POWER STATE COORDINATION INTERFACE (PSCI)

System Software on ARM® Systems

Document number: ARM DEN 0022B.b

ARM®

Power State Coordination Interface (PSCI)
System Software on ARM specification

Release information

Table 1 below lists the changes made to this document.

Table 1 Change history

Date	Issue	Confidentiality	Change
13 August 2012	A	Non-Confidential	First release
24 June 2013	B	Non-Confidential	Second release, PSCI version 0.2
25 Jun 2013	B.b	Non-Confidential	Removed inappropriate watermark, PSCI version 0.2

Table of Contents

1 Introduction

This document defines a standard interface for power management that can be used by operating system vendors for supervisory software working at different levels of privilege on an ARM device. Rich operating systems like Linux and Windows, hypervisors, secure firmware and Trusted OS implementations must interoperate when power is being managed. The aim of this standard is to ease the integration between supervisory software from different vendors working at different privilege levels.

The interface is aimed at the generalization of code in the following power management scenarios:

- Core idle management.
- Dynamic addition and removal of cores.
- big.LITTLE migration.
- System shutdown and reset.

The interface does not cover *Dynamic Voltage and Frequency Scaling* (DVFS).

The interface is designed so that it can work in conjunction with hardware discovery technologies such as *Advanced Configuration and Power Interface* (ACPI) and *Flattened Device Tree* (FDT). It is not a replacement for ACPI.

1.1 Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com, for access to ARM documentation.

1.1.1 ARM publications

The following documents contain information relevant to this document:

[1] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (ARM DDI 0406).

[2] Embedded Trace Macrocell Architecture Specification (ARM IHI 0014)

[3] Program Flow Trace Architecture Specification (ARM IHI 0035)

[4] SMC Calling Convention (ARM DEN 0028).

[5] ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (ARM DDI 0487). ARM expects this document to be available from the Infocenter from August 2013.

2 Background

Power management aware operating systems dynamically change the power states of cores, balancing the available compute capacity to the current workload, whilst striving to use the minimum amount of power. Some of these techniques dynamically switch cores on and off, or place them into quiescent states, where they no longer perform computation. This means they consume very little power. The main examples of these techniques are:

Idle Management: When the kernel in an operating system has no threads to schedule onto a core, it places that core into a clock-gated, retention, or even fully power-gated state. However the core remains available to the OS.

Hotplug: Cores are physically switched off when the compute demand is low, and then brought back online when it increases. The OS migrates all interrupts and threads away from the cores that are taken offline, and rebalances the load when they are brought back online.

big.LITTLE™ technology provides powerful options for matching compute capacity to work load by appropriately distributing the load between big and LITTLE processors. The use of processors with different compute capacities, but the same Instruction Set Architecture (ISA), provides a wider dynamic range of power and performance than is possible using designs with only one type of processor. Performance management of a big.LITTLE system intersects directly with power management. In particular, *Operating System Power Management* (OSPM) might need to turn cores and clusters on and off as it transfers computation from a big core to a LITTLE one, or from a LITTLE core to a big one.

Although it would be simpler to consider the software of an embedded system to be provided by a single vendor, in most situations this is not the case, even when the end device is delivered with fixed functionality. The ARM architecture defines a set of Exception levels [5],that support the required partitioning of the software stack used on a device. Table 2 shows this partitioning and indicates the typical vendor of each level of the stack:

Table 2 Typical partitioning of software on a system that complies with the ARM architecture

AArch32 state	AArch64 state	Stack and typical vendor
Non-secure EL0 (PL0)	Non-secure EL0	Unprivileged application, such as apps downloaded from an App Store
Non-secure EL1 (PL1)	Non-secure EL1	Rich OS kernels from for example Linux, Windows RT, iOS
Non-secure EL2 (PL2)	Non-secure EL2	Hypervisor, from vendors, such as Citrix, VMWare or OK-Labs
Secure EL0 (PL0)	Secure EL0	Trusted OS applications
Secure EL3 (PL1)	Secure EL1	Trusted OS kernels from Trusted OS vendors such as Trustonic.
Secure EL3 (PL1)	Secure EL3	Secure Monitor, executing Secure platform firmware provided by Silicon vendors and OEMs

Note: AArch32 state is the 32-bit ARMv8 execution state, and the execution state used by all ARM processors before ARMv8. In an ARMv7 processor, the Exception levels are implicit, and not identified by the processor documentation. The Virtualization Extensions provide the EL2 functionality, and the Security Extensions provide the EL3 functionality, and within a Security state the *Privilege level* (PL) identifies the Exception level hierarchy. For more information see Section 3.2 *Exception levels, the ARMv7 Privilege levels, and highest privilege*.

As various operating systems from various different vendors can be present in an ARM system, performing power control requires a method of collaboration. Considering operation in Non-secure state, if a supervisory system that is managing power, whether it is executing at the OS level (EL1) or at hypervisor level (EL2), wants to enter an idle state, power up or power down a core, perform a big.LITTLE migration, or reset or shutdown the system, supervisory systems at other levels of privilege will need to react to the power state change request.

Equally, if the power state of a core is changed by a wake up event, it might be necessary for supervisory systems running at different levels of privilege to perform actions such as restoring context. Previously there was no standard interface definition to support this interoperation and integration across the various supervisory systems. This document defines such an interface, the Power State Coordination Interface, and describes its use for idle, Hotplug, big.LITTLE management, shutdown and reset.

3 Assumptions and recommendations

This document defines an API that can be used to coordinate power control amongst the various supervisory systems concurrently running on a device. As the following sections explain, the API allows a supervisory system to request cores to be powered up or down, and to request context transfer where necessary for big.LITTLE systems. Throughout the description the document generally assumes that EL2 and EL3 are both implemented, but also covers other cases.

3.1 PSCI intended use

The PSCI:

1. Provides a generic interface that supervisory software can use to manage power in the following situations:
    a. Core idle management.
    b. Dynamic addition and removal of cores from the system, often referred to as hot plug
    c. big.LITTLE migration models, described later in this document.
    d. System shutdown and reset
2. Provides an interface that supervisory software can use in conjunction with Firmware Table (FDT and ACPI) descriptions to support the generalization of power management code.

PSCI does not cover:

1. Peripheral idle management. PSCI only applies to idle management of the cores used by the central scheduler of supervisory software.
2. Dynamic Voltage and Frequency Scaling. PSCI does not provide interfaces for the management of core clock frequencies on a device.

PSCI does not provide power state representations to supervisory software. However, it is designed so that it can also be used with hardware description technologies such as ACPI or FDT.

3.2 Exception levels, the ARMv7 Privilege levels, and highest privilege

ARMv8 introduces explicit Exception levels, that also define the software execution privilege hierarchy within a Security state. An increase in Exception level, for example from EL0 to EL1, corresponds to an increase in execution privilege.

In ARMv7, the Exception level hierarchy is implicit in the architecture:

- The Virtualization Extensions provide the EL2 functionality. This is present only in Non-secure state.
- The Security Extensions provide the EL3 functionality, including the support for two Security states. The control features of this functionality are provided by Monitor mode, that is present only in Non-secure state.

ARMv7 descriptions [1] use *Privilege levels* (PLs) to describe the software execution privilege hierarchy. Because Monitor mode was defined as a peer of the other Secure state privileged processor modes, this means the ARMv7 privilege levels are asymmetrical between Non-secure state and Secure state, as follows:

- In Non-secure state the privilege level hierarchy is:
  - PL0, unprivileged. Applies to User mode.
  - PL1, OS-level privilege. Applies to System, FIQ, IRQ, Supervisor, Abort, and Undefined modes.
  - PL2, hypervisor privilege. Applies to Hyp mode.
- In Secure state the privilege level hierarchy is:
  - Secure PL0, unprivileged. Applies only to User mode.
  - Secure PL1, Secure OS and Monitor level privilege. Applies to System, FIQ, IRQ, Supervisor, Abort, Undefined, and Monitor modes.

In the AArch32 description of execution privilege [5], which applies to both ARMv7 and ARMv8 implementations, the processor modes map to the Exception levels as follows:

- In Non-secure state the processor modes implemented at each Exception level are:
  - EL0: User mode.
  - EL1: System, FIQ, IRQ, Supervisor, Abort, and Undefined modes.
  - EL2: Hyp mode.
- In Secure state the processor modes implemented at each Exception level are:
  - Secure EL0: User mode.
  - EL3: System, FIQ, IRQ, Supervisor, Abort, Undefined, and Monitor modes.

Note: In an ARMv8 implementation, this mapping of the Secure modes applies only when EL3 is using AArch32. When EL3 is using AArch64, Monitor mode is not implemented, and if Secure EL1 is using AArch32 then Secure System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of Secure EL1. For more information see [5].

This document generally uses Exception level terminology. In the document:

- References to EL1 and EL0 mean Non-secure EL1 and EL0, unless they explicitly indicate otherwise.
- *Highest privilege* refers to the first implemented Exception level in the following sequence, that runs from highest to lowest, EL3, Secure EL1, EL2, Non-secure EL1.

3.3 Software Stacks on ARM based Systems

On a given ARM device there can be a number of supervisory software kernels or privileged software components.

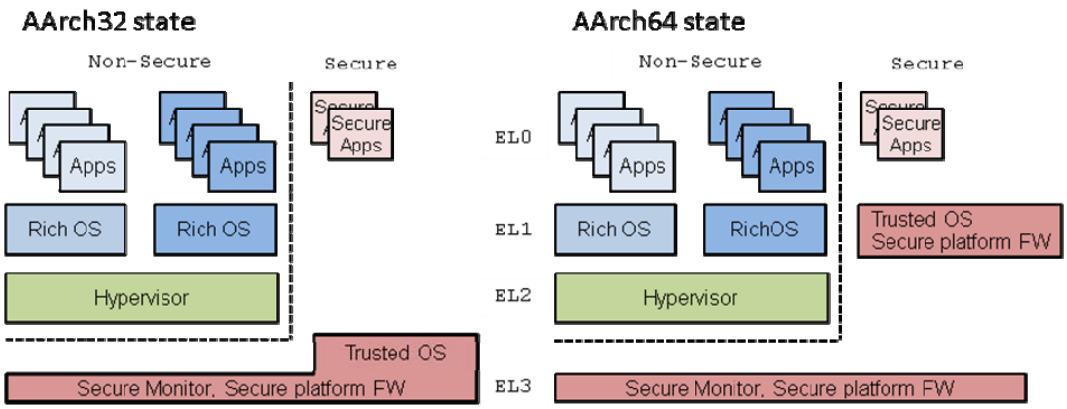

Figure 1 Software Layers of an ARM system

Figure 1 illustrates these various layers. The Normal (Non-secure) world has the following privileged components:

Rich OS kernels: such as Linux or Windows running in Non-secure EL1. When running under a hypervisor, the Rich OS kernels can be running as a guest or host depending on the hypervisor model. Section 3.6 gives more information about this.

Hypervisor: This component runs at EL2, which is always Non-secure. This component, when present and enabled, provides virtualization services to Rich OS kernels.

The Secure world has the following privileged components:

Secure Platform Firmware (SPF): Owned by the silicon vendor and OEM. On an application processor, this firmware layer must be the first thing that runs at boot time. It provides a number of services, including platform initialization, the installation of the Trusted OS, and routing of Secure Monitor Calls. Some calls are destined for the SPF and some are destined for the Trusted OS. SPF can run in EL3 and secure EL1 on ARMv8 systems using AArch64 in EL3. For ARMv7 systems, or ARMv8 systems using AArch32 at EL3, SPF executes in EL3. The implementation that acts on power management requests issued by the Power State Coordination Interface must be implemented in the SPF.

Trusted OS: This provides secure services to the Normal world, and provides a runtime environment for executing secure applications. In AArch32 state Trusted OS software executes in Secure EL3, and in AArch64 state it primarily executes in Secure EL1.

Note: The ARM Architecture Reference Manuals define two Security states, Secure and Non-secure, and ARM processor documentation uses these state names. ARM software documentation refers to Normal world and Secure world execution, to indicate that software normally executes in Non-secure state, which is not an insecure state.

The PSCI specification focuses on the interface between Secure and Normal worlds for power management. It provides a method for issuing power management requests. To deal with the requests, the SPF must include a PSCI implementation.

The PSCI specification requires communication between the SPF and a Trusted OS. Currently, how this communication is handled is specific to individual vendors. For the PSCI specification, this remains IMPLEMENTATION DEFINED, although this specification includes some recommendations.

Although the PSCI interface specification focuses on power management requests between Secure and Normal worlds, the interface can be reused easily at the junction between Rich OS kernels and hypervisors.

3.4 Conduits

The PSCI interface must support interaction at all levels of execution present on the device, where multiple levels of supervisory software might be executing. For the caller operating in the Normal world, the interface must forward a message to the Secure world. In a system that includes EL2, it must be possible to trap interface calls made by the EL1 kernel context to the hypervisor (EL2). If the hypervisor determines that a change of physical power state is required, it must then be able to use the same interface to inform the Secure world.

The conduits available to transfer a message from one Exception level to another depend on the implemented Exception levels.

The Secure Monitor Call instruction, SMC, provides a method for a Non-secure exception level to call into the Secure world. In addition, SMC execution at EL1 can be trapped by the hypervisor to EL2. This means SMC gives the flexibility required to support implementations with and without a hypervisor, making it the ideal conduit for PSCI power management requests. However, use of SMC requires the implementation of EL3 [1,5]. If EL3 is not implemented the SMC instruction is UNDEFINED and generates an exception in the calling OS.

The Hypervisor Call instruction HVC, allows a Rich OS executing at EL1 to communicate with a hypervisor at EL2. This means a hypervisor can provide a PSCI interface, using HVC as the calling conduit. If EL2 is implemented and EL3 is not implemented, then Hypervisor Call (HVC) is the only conduit available.

Note: In ARMv7, implementing EL2 requires that EL3 is also implemented [1].

If an implementation does not include either EL2 or EL3, then no coordination with other supervisory software is required, as there can only be one operating system, that must execute at EL1. Power control in this case must use board support adaptation code implemented specifically for that operating system.

Table 3 describes the different conduits available.

Table 3 Dependence of available conduits on implemented Exception levels

EL3	EL2	Conduit	Notes
✓	✓	SMC,HVC	-
✓	X	SMC	-
X	✓	HVC	EL combination not permitted in ARMv7
X	X	Platform specific code	No conduit required

Discoverability methods for the PSCI interface, and the conduit that applies, are described in Section 5.1.8.

3.5 Secure world software and power management

Most current Trusted OS implementations are not SMP-capable. When running on SMP devices, they are tied to a single core. Secure Monitor Calls destined for the Trusted OS are only expected to come from that core. The lack of SMP support in the OS helps to

keep Trusted code simple and small, which in turn aids certification. Trusted OS services are invoked from the Normal world through Rich OS drivers and/or daemons provided with the Trusted OS implementations. The threads associated with these drivers and daemons are normally affinitised to the core used by the Trusted OS.

ARM systems generally include a power controller that can control core power. It normally provides interfaces that support a number of power management functions. These often include support for transitioning cores, clusters, or a superset into low-power states. In the low-power state, the cores are either fully switched off, or in quiescent states where they are not executing code.

ARM strongly recommends the Secure world is responsible for the control of these states, using this power controller. Otherwise, the OSPM could enter a low-power mode without informing the Trusted OS. Even if such an arrangement could be made robust, it is unlikely to perform as well. In particular, for states where the core is fully power-gated, a longer boot sequence would be required on wake up, because the Secure world would require full initialization. This is because the secure components would effectively be booting from cold every time.

On a system where the Secure world is responsible for this power control, Secure components can save their state before power-off, providing a faster resumption on power-on. In addition, the Secure world might need to manage peripherals as part of a power transition.

Other forms of power management such as dynamic performance management through voltage and frequency scaling, are not covered by this interface. **ARM strongly recommends that all policy in power and performance management is performed in the Normal world**. The Normal world has greater visibility of the current use and purpose of a given device. Where the Secure world has performance requirements, it is recommended that IMPLEMENTATION DEFINED mechanisms are used to communicate those requirements to the Normal world.

3.6 Virtualization and core power policy

Hypervisors are broadly split into two basic types:

Type 1: Sometimes described as native or bare metal. Type 1 hypervisors execute directly on the hardware. Any application guest operating system sees a virtualized view of this hardware.

Type 2: Sometimes described as hosted. Type 2 hypervisors run within a host operating system. The host OS has a physical view of the hardware it runs on, but guest operating systems see a virtualized view.

This is a very broad categorization. In reality, there are variations on the above. However, in general terms these two abstractions capture the forms of power management required by the hypervisors.

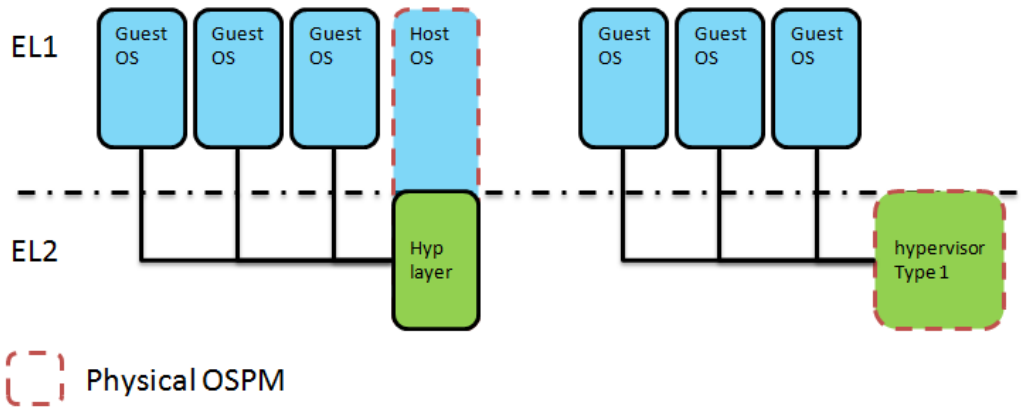

Figure 2 Power management models in virtualization

From the point of view of power management and virtualization there are two types of OSPM:

Physical OSPM: This comprises the software components that select the physical power states.

Virtual OSPM: This is an OSPM present in a guest operating system running a virtual machine, which selects virtual, rather than physical power states.

In type 2 hypervisors, the physical OSPM resides in the host. Actual power policy is controlled from a Rich OS running at EL1, rather than from the hypervisor itself. The physical OSPM is contained in this Rich OS. This layer has a physical view of the processors. This is shown on the left hand side in Figure 2.

For the power management functions covered in this document, type 2 hypervisor behavior depends on the calling OS.

If the caller is the host (type 2), the hypervisor complies with the power request, or allows the call to pass straight through to the Secure platform firmware. The hypervisor typically only performs any necessary operations resulting from the call, for example, if necessary, saving state on a power down for registers the host cannot access. Thereafter, the hypervisor calls through to the Secure platform firmware using the parameters supplied by the caller. If no special operations are required, the hypervisor does not even trap calls from the host, and instead routes them directly to the Secure platform firmware.

When a type 2 hypervisor is implemented, guests use a virtual OSPM. They can also issue power requests through the PSCI interface APIs, but the requests are issued in relation to virtual cores, and virtual power states. These requests are trapped by the hypervisor which issues them back to the physical OSPM. This component can then determine whether physical power management is required as a result. For these guests the power calls effectively terminate at the hypervisor.

With a type 1 hypervisor, power policy is typically owned entirely by the hypervisor. This is shown on the right hand side in Figure 2. The physical OSPM is implemented in the hypervisor. In this case all guests have a virtualized view of the cores. The hypervisor determines from the virtual power states of the guests whether physical power control is required, and if so uses the PSCI interface API to coordinate with the Secure platform firmware. Guests can also use this API, to communicate virtual power requirements to the hypervisor. For these guests the calls effectively terminate at the hypervisor.

In some cases type 1 hypervisors delegate power management to a privileged guest. In this case the physical OSPM is implemented in that privileged guest. This power management approach is equivalent to the model described above for type 2 hypervisors, as shown in the left hand side of Figure 2.

4 Requirements

4.1 Idle management

When a processor core is idle the OSPM transitions it into a low power state. Typically, a choice of states is available, with different entry and exit latencies, and different levels of power consumption, associated with each state. The state that is used typically depends on how quickly the core will be needed again. The power states that can be used at any one time might also depend on the activity of other components in an SoC, beside the cores. Each state is defined by the set of components that will be clock-gated or power-gated when the state is entered. States are sometimes described as being shallow or deep. Typically a state X is said to be deeper than a state Y if:

- The set of components that are powered-down in state X subsumes and is larger than the corresponding set for state Y.
- If the set of components is the same but various power modes are supported, the modes used in state X save more power than those used in state Y.

The time required to move from a low power state to a running state, known as the wakeup latency, is longer in deeper states.

Although idle power management is driven by thread behavior on a core, the OSPM can place the platform into states that affect many other components beyond the core itself. If the last core in a cluster becomes idle, the OSPM can target power states that affect the whole cluster. Equally, if the last core in a SoC goes idle the OSPM can target power states that affect the whole SoC. The choice is also driven by the usage of other components in the system. A typical example is placing memory in self-refresh when all cores, and any other bus masters, are idle. The OSPM has to provide the necessary power management software infrastructure to determine the correct choice of state.

In idle management, once a core or cluster has been placed into a low power state, it can be reactivated at any time by a processor wakeup event. That is, an event that may wake up a processor from a low power state, such as interrupt. No explicit command is required by the OSPM to bring the core or cluster back into operation. The OSPM considers the affected core or cores to be available at all times even if they are currently in a low power state.

An ARM core can be in any of the following power states:

Run: The core is powered up and operational

Standby: The core is powered on, but measures are employed to reduce energy consumption. In a typical implementation, the core enters standby by executing a `WFI` or `WFE` instruction and exits on a corresponding wake-up event. The core preserves all core state. Changing from standby to running operation does not require a reset of the processor. In standby, all core context is maintained, and can be directly accessed on wake up. No special actions are required by the operating system to ensure context is maintained. An external debugger can access debug registers in the core power domain [1,5].

Retention: The core state, including the debug settings, is preserved in low-power structures, allowing the core to be at least partially turned off. Changing from low-power retention to running operation does not require a reset of the core. The saved core state is restored on changing from low-power retention state to running operation. From an operating system point of view there is no difference between a retention state and standby state, other than method of entry, latency and usage-related constraints. However, from an external debugger point of view the states differ as External Debug Request debug events stay pending and debug registers in the core power domain cannot be accessed [1,5].

Power-down: In this state the core is powered off. Software on the device needs to save all core state, so that it can be preserved over the power-down. Changing from power-down to running operation must include:

- A reset of the core, after the power level has been restored.
- Restoring the saved core state.

The defining characteristic of power down states is that they are destructive of context. This affects all the components that are switched off in a given state, including the core, and in deeper states other components of the system such as the GIC or platform-specific IP. Depending on how debug and trace power domains are organized, in some power-down states one or both of debug and trace context might be lost. Mechanisms must be provided to enable the operating system to perform the relevant context saving and restoring for each given state. Resumption of execution starts at the reset vector, after which each OS must restore its context.

To an operating system managing power, a standby state is mostly indistinguishable from a retention state. The difference is evident to an external debugger, and in hardware implementation, but not evident to the idle management subsystem of an operating system. Consequently, unless otherwise stated, this document uses the term standby to refer to both standby and retention states.

An interface is required so that the OSPM can place a core into a low power state when it has no work for it. The messages sent through this interface must be received by all relevant levels of execution. That is, if EL2 and EL3 are implemented, a message sent by a Rich OS should be received by a hypervisor and the Secure world. Within the Secure world the message needs to be seen by a Trusted OS, if present, and by any Secure platform firmware. This means each level of supervisory software can determine whether it must perform context saving.

ARM expects that the Exception level with the highest privilege, as defined in Section 3.2, will be the component that can program the power controller to enter an idle state. Typically, this will be the Secure platform firmware. Therefore, the PSCI interface requires a mechanism for the OSPM to pass the desired idle state to the next exception level.

For standby states, that do not require any explicit programming of the power controller, no specific interface is required. The OSPM can use `WFI` or `WFE` instructions directly. However, for deeper standby states that require programming a power controller, it is advantageous to provide an interface that can hide the platform-specific code that accesses the power controller.

Power-down states require an interface so that each level of execution can save and restore its context appropriately. For power down states, the interface requires a return address. This is the address at which the calling OS expects resumption of execution on wakeup at its Exception level. From a powered-down state, the core restarts at the reset vector, in Secure state if the implementation includes EL3. After initializing, the Secure world must re-start execution of the OS that called the power down interface, at the required return address.

4.2 CPU hotplug and secondary CPU boot

CPU hotplug is a technique that can dynamically switch processors on or off. Hotplug can be used by the OSPM to change available compute capacity based on current compute requirements. Hotplug is also sometimes used for reliability reasons. There are a number of differences between hotplug and use of a power-down state for idle:

1) When a core is hot unplugged, the supervisory software stops all use of that core in interrupt and thread processing. The calling supervisory software regards the core as no longer available.

2) The OSPM has to issue an explicit command to bring a core back online, that is, hotplug a core. The appropriate supervisory software will only start scheduling on or allowing interrupts to that core after this command.

3) With hotplug, wake-up events that could restart a powered-down core, are not expected on the cores that have been hot unplugged. It is an error for a core to execute in an Exception level that has previously hot unplugged this core.

Operating systems typically perform much of the kernel boot process on one primary processor core, bringing secondary processors online at a later stage. For systems that support hotplug, the operations involved in booting a processor for secondary boot, or hotplug, are the same. Therefore they can be provided by a single interface. This interface requires the following properties:

1) The OSPM can request that a core be powered up. The OSPM must provide an appropriate start address for its Exception level. The provision of a start address means the receiver can shortcut the bootloader when onlining a core in the calling OS. Different addresses can be provided to distinguish between different startup reasons. Alternatively, the OSPM can use internal per-core data structures for this purpose.

2) The OSPM can request powering down the core, and inform higher Exception levels it is doing so.

4.3 big.LITTLE

There are currently three scheduling models being considered for big.LITTLE systems:

Cluster migration: In Cluster migration, the big.LITTLE hardware platform is typically defined by an SoC that has the same number of big and LITTLE cores. Only one cluster, the big or the LITTLE, is active at any one time. The OSPM considers the overall performance loading on the currently-active cluster. Typically, this is the load of the core that is busiest in the cluster. If the load warrants a change from big to LITTLE or from LITTLE to big, the OSPM synchronizes all the cores and then transfers all compute context to the other cluster. As part of this process every operating system, on every core, has to save its state whilst still executing on the old (outbound) cluster. When the new (inbound) cluster boots up, the operating systems need to restore their state on each core. Once the inbound cluster has restored context, the outbound cluster is switched off. In this scheme operating systems maintain a logical view of the compute cluster and the cores it contains, where the logical cluster can at any one time be instantiated as the big cluster or the LITTLE cluster.

CPU migration: In CPU migration, each core on the LITTLE cluster is paired with a core on the big cluster. The OSPM monitors the load on each core. High load causes the execution context to be moved to the big core, and equally when the load is low, the execution is moved to the LITTLE core. Only one core in the pairing is active at any time. When the load is transitioned from an outbound core to an inbound core the former is switched off. This model allows a mix of big and LITTLE cores to be active at any one time. Much like cluster migration, in this scheme operating systems maintain a logical view of processor cores, where the logical core can at any one time be instantiated as a big core or a LITTLE core.

MP: The operating system is topology aware and operates across all cores in all clusters. The OS is aware of the compute capacity differences between big and LITTLE cores. The scheduler in the OS assigns tasks to the right type of core based on the compute requirements of each task. OSPM idle management and hotplug powers-off unused, or under-utilized cores.

Migration approaches require explicit power control of cores as part of the migration process. When the compute is transferred from the outbound core or cluster to its inbound

sibling, the former can be powered down. In MP, power-down only occurs through hotplug or idle management. Migration adds another decision point in the OS that can power-down cores, in addition to idle and hotplug decisions.

In addition to power interfaces, big.LITTLE migration models require an interface that an operating system can use to inform higher Exception levels that migration is required. This allows operating systems to move context from one core to another.

4.4 Save and Restore

Use of power down states in idle management, where context is lost, and use of big.LITTLE migration, both require the ability to resume execution on a previously powered down core. The return to execution must be transparent to the applications that were executing on the core before its shutdown. To achieve this, operating systems must save the context of execution for the core. When a core resumes its context must be restored so that execution can restart from the same point with the same context.

To implement this, every operating system must provide a software stack that can:

- Save the architectural context that will be lost when power is removed.
- Save the non-architectural state that will be lost when power is removed. There might be SoC implementation-specific components that have a state that needs to be saved.
- In a big.LITTLE migration, interrupts must be redirected from each outbound core to the corresponding inbound core.
- Provide a return path for execution can perform basic initialization and the restoration of non-architectural and architectural state.

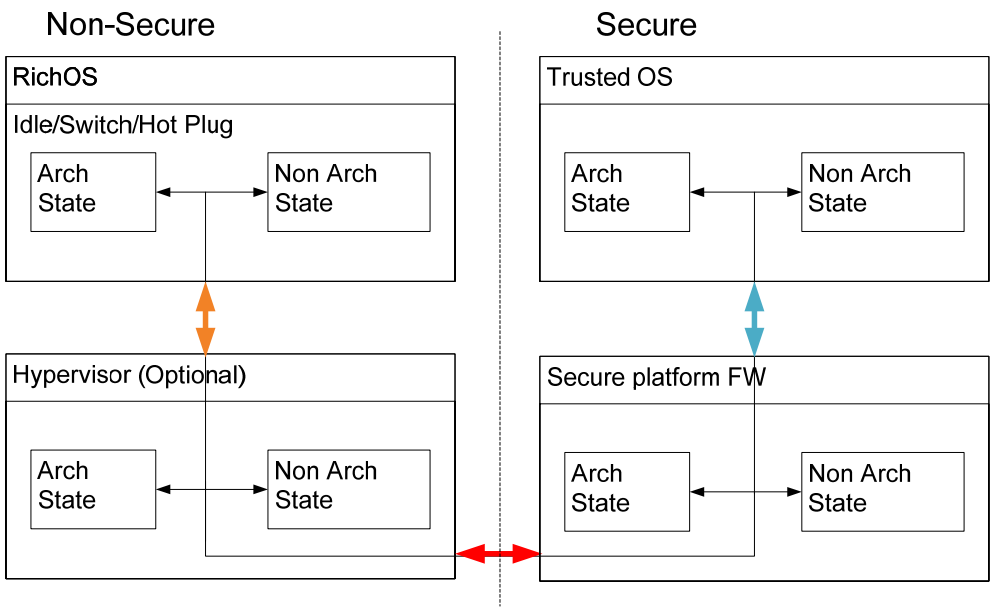

Figure 3 Save and restoring context across operating systems

Figure 3 depicts the components involved in such a stack across all levels of supervisory software that might be executing on an ARM system. The PSCI specification focuses primarily on the interface between the Normal world and Secure world, shown by the red arrow in Figure 3. It also covers the interface between rich operating systems and hypervisors, shown by the orange arrow. This specification does not address the interface between the Secure platform and a Trusted OS.

Every OS must save and manage its architectural state and the GIC state. The former includes:

- The general-purpose register bank.
- The NEON and floating-point register bank.
- System control registers, for example, MMU, Generic Timer at the level appropriate to the OS.
- Debug state.
- Trace state.

In big.LITTLE migration models, for a GICv2 system, management of interrupts involves:

1) Saving the CPU interface registers (binary point and priority mask and control register) for any outbound core.
2) Retargeting shared peripheral interrupts (SPIs) to the CPU interfaces of any inbound core, and away from the corresponding outbound core.
3) The OSPM must update its internal representations to ensure any future software generated interrupts (SGIs) are sent to any inbound core instead of the corresponding outbound core. Any pending SGIs must be cleared on the outbound core and pended on the inbound core.
4) The OSPM must to prevent any new private peripheral interrupts (PPIs) from arising on any inbound core before initiating the migration. Any pending PPIs must be cleared on any outbound core and pended on the corresponding inbound core. After the switch PPI sources must be re-enabled.
5) Transfer state of distributor banked registers (group, set-enable/clear-enable for PPIs, priority, non-secure access and configuration if implementation requires it), from any outbound core to the corresponding inbound core.

In core hotplug OFF it is also necessary to manage the following GIC state:

1) The OSPM must remove the core from its representation of available cores.
2) This means the OSPM must ensure that no SGIs and no PPIs are pending on a core that is going to be hotplugged off.
3) The OSPM must retarget SPIs away from a core before hotplugging it off.

In core hotplug ON:

1) The OSPM can instruct the kernel to add the core to its representation of available cores.
2) If required, the OSPM can retarget SPIs onto the newly-available core.

In a typical system, before powering down a core as part of idle management or hot unplug, the CPU interface on that core must be placed in a quiescent state. This ensures that the final `WFI` instruction, that signals to a power controller that the core can be powered down, will not complete even if an interrupt is pending.

When powering down a core for idle management it might be necessary to save the CPU interface state and to save GIC distributor banked (per core) registers. This includes group, set-enable and clear-enable for PPIs, priority, Non-secure access and configuration if the implementation requires it. In addition, if entering a system power down state where the GIC state will be lost, the last core to power down must save shared GIC registers, meaning it must save the control, enable, target, configuration, group and Non-secure access registers.

4.4.1 Debug and Trace save and restore

The ARM architecture provides support for external debug, and for self-hosted debug. External debuggers provide hardware assisted debug that can modify core and memory data. The external debugger can also modify debug context to manage debug events such as breakpoints and watchpoints. Finally, an external debugger can also affect the power state of a core by acting as a source of wakeup events, or by preventing power downs. For more information see [1], [5].

Saving and restoring invasive debug context

Supporting for debug on multicore systems is complicated by the fact that system software can move the software threads being debugged from one core to another. Therefore, supporting debug requires a strategy that either prevents this migration, or that appropriately duplicates debug context on every core. Duplicate programming provides a more realistic debug method from a scheduling point of view, as there is no need for pinning.

Regardless of the method chosen, when using an external debugger, the debugger might interact with a core that is powered-down. Any attempts by the debugger to access a debug register when the core power domain is powered-down, or in a low power state where the core power domain registers cannot be accessed, will return an error. The debugger can retry the access. However, if the core power domain is regularly put into such a state, this might lead to unreliable debugger behavior. In a multicore system utilizing dynamic power control, it might be rare for all the cores to be powered up simultaneously, but this is necessary for duplicating programming across all cores. There are two options an external debugger can use in this case:

1. A debugger can request emulation of power down, through the use of **DBGNOPWRDWN** and **DBGPWRUPREQ,** as documented in [1,5]. This approach prevents use of debug over a power down.
2. If the supervisory software supports debug over power down, the debugger can allow full power cycling of cores. Under this scheme the debugger keeps a valid central copy of the debug context. When a user changes this context, the debugger must change the context of the cores to match. If any core is powered down, then debugger can use the OS Unlock Catch debug event to halt the core after it has woken up, and completed it debug context restore sequence. At this point the debugger can program the debug registers from Debug state. This method means the debugger can avoid using emulation of power down.

The second option requires a method the debugger can use to inform the supervisory software that it must save and restore debug registers during a power down transition. ARM recommends the use of CLAIM tag registers for this purpose. CLAIM tags can also be used to prevent conflicts between external debug and self hosted debug use of debug registers. The following CLAIM tags bits should be used for controlling save and restore of debug context:

DBGCLAIM[0] should be set to 1 to indicate that debug is in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software must perform a save or restore sequence of the debug context
2. Debug context will not be overwritten by self hosted debug software

DBGCLAIM[1] should be set to 1 to indicate that debug is in use by a self hosted debugger. This bit can be used to indicate to power management software that it must save and restore debug context when power cycling.

In an AArch32 implementation that is using v7 Debug, **DBGCLAIM[6]** should be set to 1 by the supervisory software as an acknowledgement for bit [0] (only required for compatibility with v7 Debug).

These claim tags can be used to indicate that debug context requires a save/restore sequence.

Note: For AArch32 implementation that use v7 Debug, ARM recommends that **DBGCLAIM**[6] is reserved for the operating system to acknowledge that it will perform save/restore. The external debugger can write to **DBGCLAIM** when the core is powered down, meaning it should either not clear **DBGPRCR**.**CORENPDRQ** or not program any volatile state until the power-down software acknowledges that it will save/restore the state by setting **DBGCLAIM**[6] to 1.

Debug CLAIM tags should be programmed by the external debugger, or self hosted debug agent using the following sequence

1. Check CLAIM tags, if ownership is clear (bit [0:1]) proceed otherwise fail, or retry later
2. Claim ownership by setting bit 0, for external debugger, or bit 1 for self hosted debug agent
3. Check CLAIM tags again:
    a. If only the bit programmed in step 2 is set, and the other is clear, then claim tags have been successfully programmed.
    b. Otherwise, if both bits are set, then clear bit set in step 2 and fail or retry later.

Self hosted debug does not always require a save and restore sequence of debug context used in invasive debug to support power management transitions. Breakpoint and watchpoint registers are associated with threads, and these can be migrated from one core to another through OS scheduling. Therefore, the operating system might already implicitly save and restore breakpoint and watchpoint data with every context switch. However, generic events that are not associated with a specific thread must be saved and restored for power-down states, using the sequences described in [1], [5].

With self hosted debug, big.LITTLE migration models must save and restore debug context as part of a normal context migration. If the migration always occurs in specific software threads, and there is no need to debug these threads, then only generic events, not associated with threads, and not saved/restored through scheduling, must be saved on the outbound core and restored on the inbound core. If the migration does not occur in specific software threads, or support for debug of the migration threads is required, then all debug context, used by the self hosted debugger, must be saved and restored.

Saving and restoring Performance Monitoring Units

*Performance Monitoring Unit*s (PMUs) also require some management on MP systems. Once again external and self hosted debuggers must ensure that they do not conflict with the use of PMUs. In addition, the external debugger might require save and restore of PMU registers during a power transition. ARM recommends the use of CLAIM tag bits to indicate whether PMUs are in use by the external debugger or self hosted debug as follows:

DBGCLAIM[2] should be set to 1 to indicate that PMUs are in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software will perform a save/restore sequence of the PMU context.
2. PMU context will not be overwritten by self hosted debug software.

DBGCLAIM[3] should be set to 1 to indicate that PMUs are in use by a self hosted debugger. This bit can be used to indicate to power management software that it must save and restore debug context over a power cycle.

In an AArch32 implementation that is using v7 Debug, **DBGCLAIM[7]** should be set to 1 by the supervisory software as an acknowledgement for bit [2]. For more information see the Note for **DBGCLAIM[1]** .

Supervisory software performing big.LITTLE migration can take different approaches to PMUs. For events that are common to big and LITTLE processors, it is possible to provide a logical view of cores. Under this scheme the user sees events for a single core, which can at any time be instantiated as a big processor or a LITTLE processor, however the event counting and event interrupts are treated as coming from the single logical core. This would require the migration stack to transfer PMU state from one core to another as it moves the rest of the context.

Alternatively, the supervisory software could just provide a physical view of cores. This does not require transferring any context, but the programmer needs to be aware that events are counted separately between the LITTLE and big cores. In this case the supervisory software must save the PMU context of each core when it is powered down, and restore it when powered up, without migrating data. A big.LITTLE MP solution, or any other multicore operating system solution only needs to support physical views of core. When working with heterogeneous systems it is also worth noting that:

- There will be events that are specific to each core type.
- The number of counters can differ between cores.
- Even events that are common to different types of core might require specific interpretation, due to the differences in microarchitecture.

ARM recommends always providing a physical view, as appropriate interpretation of PMU requires understanding of the microarchitecture of the devices under test.

Saving and restoring trace state

Using a similar handshake to that proposed for debug state, ARM recommends the use of ETMCLAIM registers to indicate whether an external or a self hosted debugger is using trace, and to indicate the need to save and restore:

ETMCLAIM[0] should be set to 1 to indicate that trace is in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software will perform a save/restore sequence of the trace context.

2. Trace context will not be overwritten by self hosted debug software.

ETMCLAIM [1] should be set to indicate that debug is in use by self hosted debugger. And the self hosted debugger expects the debug registers.

In an AArch32 implementation that is using ETMv3.4 or PFTv1.0,**ETMCLAIM [6]** should be set to 1 by the supervisory software as an acknowledgement for bit [0].

The save and restore sequence must be appropriate for the type of trace architecture implemented by the core, and this might differ depending on core type. For example the Cortex-A15 processor uses Program Flow Trace Architecture [3], whereas the Cortex-A7 processor uses the Embedded Trace Macrocell Architecture [2]. Both architectures support ETMCLAIM tags.

Saving and restoring debug context and Exception levels

Save and restore sequences for debug and trace state can be performed at any privileged Exception level that is making use of the services. It might be that virtualized access is being provided to debug services, by higher Exception levels. In this case the higher Exception levels must trap accesses as appropriate.

In all cases the save sequence must use the CLAIM tags to decide whether a save and restore sequence is required, and whether it needs to include the subset of registers affected by self hosted debug, or the set affected by external debug. For descriptions of the

save and restore sequences see the ARMv7 ARM [1], or the ARMv8 ARM [5], as well as Program Flow Trace Architecture Specification [3], or the Embedded Trace Macrocell Architecture Specification [2].

4.5 Shutdown and reset

Just as there is no common way to request a power down of a core on ARM based systems, there is no standard way to request a system shutdown or system reset. Therefore, PSCI interface provides calls to request these system functions. This allows a silicon vendor to provide a common implementation of these functions that is independent of the supervisory software running on the device.

5 Functions

This section introduces the functions for Power State Coordination.

The APIs are described here without reference to the underlying conduit (SMC or HVC). However, the functions adhere to the SMC calling conventions [4]. In an implementation that includes EL2 but not EL3, a hypervisor providing support to a PSCI compliant EL1 rich OS, can use HVC as the conduit. In this case it must use the same parameters.

PSCI functions can only be called from the Normal world (EL1 or EL2).

5.1 Function prototypes

5.1.1 PSCI_VERSION

PSCI_VERSION	Return the version of PSCI implemented
Parameter	
uint32 Function ID:	0x8400 0000
Return	
uint32	*Bits [31:16] Major Version: must be 0* *Bits [15:0] Minor Version: must be 2*

5.1.2 CPU_SUSPEND

CPU_SUSPEND	Suspend execution on a core. This call is intended for use in idle subsystems where the core is expected to return to execution through a wake-up event.
Parameter	
uint32 Function ID:	0x8400 0001 AArch32 version 0xC400 0001 AArch64 version
uint32 power_state	Bits [0:15] StateID: platform specific state ID: This is platform specific, the number is understood by the firmware, and used to program the power controller. Bit [16] StateType: 0: Standby state 1: Power-down state Bits [17:23] : Must be zero Bits [24:25] AffinityLevel: Target affinity level. Describes whether the state affects a processor or a cluster. The level of affinity that will be powered down includes the current processor. See Section 5.4.1 for more details. Bits [26:31] : Must be zero

uint32/uint64 entry_point_address	This parameter is only valid if the target power state is a Power-down state. For Standby states the value passed in is ignored. If caller is AArch64 then this is 64 bit entry point physical address or intermediate physical address (IPA). If caller is AArch32 then this is 32-bit entry point physical address (or IPA). Note that entry point addresses requiring more than 32 bits are not supported for an AArch32 caller, regardless of whether the caller supports addresses larger than 32-bit. **Note**: In an ARMv7 implementation, this means the 32-bit limit applies even when the implementation includes the Large Physical Address Extension. Address at which the core must resume execution, when it enters the exception level of the caller, following wake up from a power down state.
uint32/uint64 context_id	This parameter is only valid if the target power state is a Power-down state. For Standby states the value is ignored. If caller is AArch64 then this is 64-bit value. When the core identified by `target_cpu`, following wake-up from a Power-down state, first enters the Exception level of the caller, this value must be present in X0. If the caller is AArch32 then this is a 32-bit value. When the core identified by `target_cpu`, following wake-up from a Power-down state, first enters the Exception level of the caller, this value must be present in R0.
Return	

int32	`SUCCESS`
	`INVALID_PARAMETERS`
	(See Section 5.2.2 for return values and error codes)

5.1.3 CPU_OFF

CPU_OFF	Power down the calling core. This call is intended for use in hotplug. A core that is powered down by `CPU_OFF` can only be powered up again in response to a `CPU_ON`.
Parameter	
uint32 Function ID:	0x8400 0002
Return	
int32	The call does not return when successful. Otherwise: DENIED (See Section 5.2.2 for return values and error codes)

5.1.4 CPU_ON

CPU_ON	Power up a core. This call is used to power up cores that either: • Have not been booted yet into the calling supervisory software. • Have been previously powered down with a `CPU_OFF` call.
Parameter	
uint32 Function ID:	0x8400 0003 AArch32 version 0xC400 0003 AArch64 version
uint32/uint64 target_cpu	This fields follows the MPIDR formatting of ARM architecture[1,5]. If the calling Exception levels is using AArch32, the format is: Bits [24:31] Must be zero Bits [16:23] Aff2 : Match Aff2 of target processor MPIDR Bits [8:15] Aff1 : Match Aff1 of target processor MPIDR Bits [0:7] Aff0 : Match Aff0 of target processor MPIDR If the calling Exception levels is using AArch64, the format is: Bits [40:63]: Must be zero Bits [32:39] Aff3 : Match Aff3 of target processor MPIDR Bits [24:31] Must be zero Bits [16:23] Aff2 : Match Aff2 of target processor MPIDR Bits [8:15] Aff1 : Match Aff1 of target processor MPIDR Bits [0:7] Aff0 : Match Aff0 of target processor MPIDR
uint32/uint64 entry_point_address	If caller is AArch64 then this is 64-bit entry point physical address or intermediate physical address (IPA). If caller is AArch32 then this is 32-bit entry point physical address (or IPA) Address at which the processor must commence execution, when it enters the Exception level of the caller.
uint32/uint64 context_id	If caller is AArch64 then this is 64-bit value. When the core identified by `target_cpu` first enters the Exception level of the caller, this value must be present in X0. If caller is AArch32 then this is 32-bit value. When the core identified by `target_cpu` first enters the Exception level of the caller, this value must be present in R0.
Return	

int32	`SUCCESS`
	`INVALID_PARAMETERS`
	`ALREADY_ON`
	`ON_PENDING`
	`INTERNAL_FAILURE`
	(see Section 5.2.2 for return values and error codes)

5.1.5 AFFINITY_INFO

AFFINITY_INFO	Enables the caller to request status of an affinity level.
Parameter	
uint32 Function ID:	0x8400 0004 AArch32 version 0xC400 0004 AArch64 version
uint32/uint64 target_affinity	uint32 if called from an Exception level that is using AArch32, and uint64 if called from an Exception level that is using AArch64. This follows the same format as the `target_cpu` parameter of a `CPU_ON` call (see Section 5.1.4)
uint32 lowest_affinity_level	Denotes the lowest affinity level field that is valid in target_affinity parameter. This argument allows the caller of `AFFINITY_INFO` to request information about affinity levels higher than 0. Possible values are: 0: All affinity level fields in `target_affinity` are valid. In a system where processors are not hardware threaded, the `target_affinity` parameter would represent an individual core. 1: Aff0 field should be ignored. The `target_affinity` parameter denotes an affinity level 1 processing unit. 2: Aff0 and Aff1 fields should be ignored. The `target_affinity` parameter denotes an affinity level 2 processing unit. 3: Aff0, Aff1 and Aff2 fields should be ignored. The `target_affinity` parameter denotes an affinity level 3 processing unit. This matches the format of the `AffinityLevel` field in the `CPU_SUSPEND power_state` parameter (see Section 5.4.1)
Return	

int32	2 `ON_PENDING`: the affinity level is transitioning to an `ON` state
	1 `OFF`
	0 `ON`: At least one core in the affinity level is `ON`
	`INVALID_PARAMETERS`
	`NOT_PRESENT`
	`DISABLED`
	(see Section 5.2.2 for return values and error codes)

5.1.6 MIGRATE

MIGRATE	This is used to ask a uniprocessor Trusted OS to migrate its context to a specific core.
Parameter	
uint32 Function ID:	0x8400 0005 AArch32 version 0xC400 0005 AArch64 version

uint32 target_cpu	This field follows the same format as `target_cpu` parameter of a `CPU_ON` call (see Section 5.1.4)
Return	

int32	`SUCCESS`
	`NOT_SUPPORTED`
	`INVALID_PARAMETERS`
	`DENIED`
	`INTERNAL_FAILURE`
	`NOT_PRESENT`
	(see Section 5.2.2 for return values and error codes)

5.1.7 MIGRATE_INFO_TYPE

MIGRATE_INFO_TYPE	This function allows a caller to identify the level of multicore support present in the Trusted OS.
Parameter	
uint32 Function ID:	0x8400 0006
Return	
const uint32	0 Uniprocessor migrate capable Trusted OS. The Trusted OS will only run on one core. The Trusted OS supports the `MIGRATE` function and be migrated to any core that has not been turned off with `CPU_OFF`. Attempts to call `CPU_OFF` on the core where the Trusted OS is resident, will fail with `DENIED`(-3). 1 Uniprocessor not migrate capable Trusted OS. The Trusted OS will only run one core. The Trusted OS does not support the `MIGRATE` function . Calls to `MIGRATE` will fail with `DENIED`(-3). Attempts to call `CPU_OFF` on the core where the Trusted OS is resident will fail also with `DENIED`(-3). 2 Trusted OS is either not present or does not require migration. A system of this type does not require the caller to use the `MIGRATE` function. `MIGRATE` function calls return `NOT_SUPPORTED`(-1)

5.1.8 MIGRATE_INFO_UP_CPU

MIGRATE_INFO_UP_CPU	For a uniprocessor Trusted OS, this function returns the current resident core
Parameter	
uint32 Function ID:	0x8400 0007 32 bit version 0xC400 0007 64 bit version
Return	

uint32/uint64	UNDEFINED if `MIGRATE_INFO_TYPE` returns 2 MPIDR based value of processor where the Trusted OS is resident if return value of `MIGRATE_INFO_TYPE` is 0 or 1. This field follows the same format as `target_cpu` parameter of a `CPU_ON` call (see Section 5.1.4).

5.1.9 SYSTEM_OFF

SYSTEM_OFF	Shutdown the system To the caller, the behavior is equivalent to a hardware power down. For a guest OS this will be virtualized, because it is running on a virtual machine. If there is no hypervisor present, or if a hypervisor calls `SYSTEM_OFF`, the whole system must be shutdown. Only a physical action by a user can bring the system into operation after this call. The following start will be a cold boot.
Parameter	
uint32 Function ID:	0x8400 0008
Return	
void	Does not return

5.1.10 SYSTEM_RESET

SYSTEM_RESET	Reset the system.
Parameter	
uint32 Function ID:	0x8400 0009
Return	
void	Does not return

5.2 Arguments and return values in PSCI

5.2.1 Register usage in arguments and return values

The SMC calling convention [4], provides support for calls using only 32-bit parameters (SMC32), and for calls using 32 and 64-bit parameters (SMC64). Some of the PSCI functions defined above use only SMC32, some of the functions have both an SMC32 and an SMC64 version.

For PSCI functions that use only 32-bit parameters, the arguments are passed in R0 to R3 (AArch32) or W0 to W3 (AArch64), with return values in R0 or W0.

For versions using 64-bit parameters, the arguments are passed in X0 to X3, with return values in X0/W0 (depending on the return parameter size).

In line with the SMC calling convention, the immediate value used with an SMC instruction must be 0.

ARM recommends that a hypervisor providing a PSCI functions through an HVC conduit uses the same approach to the immediate value and parameter passing. This aids the generalization of client code.

5.2.2 Return Error codes

Table 4 defines the values for error code used with PSCI functions:

Table 4 Return error codes

SUCCESS	0
NOT_SUPPORTED	-1
INVALID_PARAMETERS	-2
DENIED	-3
ALREADY_ON	-4
ON_PENDING	-5
INTERNAL_FAILURE	-6
NOT_PRESENT	-7
DISABLED	-8

5.3 PSCI_VERSION

5.3.1 Intended Use

A caller can use the PSCI_VERSION function to ascertain the current version of the interface. The version number is a 32-bit unsigned integer, with the upper sixteen bits denoting the major revision, and the lower sixteen bits denoting the minor revision.

Once the major revision reaches 1 the following rules apply to the revision numbering:

- Different major revision values indicate possibly incompatible functions.
- For two revisions, A and B, for which the major revision values are identical, if the minor revision value of revision B is greater than the minor revision value of revision A, then every function in revision A must work, with a compatible effect, in revision B. However it is possible for revision B to have a higher function count than revision A.

Note: To reflect the fact that this specification is still under review by ARM and its partners, the current major revision number is 0. During this time incompatible changes may be made even if only the minor revision changes.

For more detail of the differences between this version and the first draft proposal of the PSCI specification see Section 6.

5.3.2 Implementation responsibilities

An implementation conforming to the PSCI specification must implement and support all of the functions described except for `MIGRATE` which is optional. This does not mean that supervisory software using the implementation is required to use all of the functions implemented.

Note: An implementation conforming to the specification described in issue B of this specification must return a minor version number of 2 and major version number of 0.

5.4 CPU_SUSPEND

5.4.1 Intended use

The `CPU_SUSPEND` API is used to move a core, or cluster, into a low power state. The function indicates that the caller intends to make use of the core in the future, but that it has no current work for it. The `CPU_SUSPEND` API is called by an OSPM as part of idle management.

The `power_state` parameter describes the type of state that is required. The bit fields in this parameter that describe the state are:

AffinityLevel

Describes the level of affinity to be powered down (core, cluster, group of clusters). It is only possible to call `CPU_SUSPEND` from the current core. That is, it is not possible to request suspension of another core. The `AffinityLevel` field for the `power_state` uses a scheme based on the recommended use of MPIDR provided by the ARM ARM[1]. The values of `AffinityLevel` can range from 0 to 3. In a typical system, following the recommendations in the ARM ARM [1], the affinity levels are defined as shown in Table 5..

Table 5 AffinityLevel parameter

Core type	AffinityLevel	Description
Single hardware thread	0	Current core
	1	Current cluster
	2	Group of affinity level 1 clusters as defined in row above
	3	Group of affinity level 2 clusters as defined in row above
Multiple hardware threads	0	Current hardware thread in a multithreaded core
	1	Current core
	2	Current cluster
	3	Group of affinity level 2 clusters as defined in row above

The `lowest_affinity_level` affinity level parameter of the `AFFINITY_INFO` uses the same format as `AffinityLevel`.

StateType

The parameter also covers the type of state and whether it is a power down, where context will be lost, or a standby state, which retains context.

StateID

Field to express a platform-specific state ID. Generally, standby states do not require an OS to use the PSCI `CPU_SUSPEND` function. Supervisory software can enter a standby state by executing a `WFI` (Wait For Interrupt) or `WFE` (Wait For Event) instruction, without having to call a higher Exception level. However, many platforms implement a number of retention states, which to the caller look largely indistinguishable to a standby state. These states require platform-specific programming. PSCI provides a method for abstracting access to these states through the platform-specific state ID, that provides a way to identify them. Some platforms might also implement special cache power states that, for example, retain memory, and therefore require custom programming. The ID might also be used to express power down states that do not require cache cleaning.

A given platform will support a fixed set of states for each processor, cluster or overall system. These will give rise to a set of valid `power_state` values. ARM recommends that these states are expressed through firmware tables to OSPMs. This aids generalization of OSPM code.

The semantic expressed by the caller through the power state parameter is not a mandatory requirement to enter the specific state. Instead, the power state parameter indicates the deepest state the caller can tolerate. The PSCI implementation is expected to coordinate requests coming from all cores to determine the deepest state that the set of cores can tolerate. For the purposes of this coordination, a core that has not been powered up through a call to `CPU_ON`, or that has been powered down through a call to `CPU_OFF`, is assumed to have requested the deepest platform state available.

The entry point address parameter is used by the caller to express where code execution needs to resume at wake-up time. The parameter must be a physical address (PA), or, for a guest OS in a virtualized platform, an intermediate physical address (IPA). In the guest OS case the hypervisor must trap the call, so that it can convert the IPA to a PA.

The context identifier parameter only meaningful to the caller. The PSCI implementation must preserve a copy of the value passed in this parameter. Following wake-up from a power down state, the PSCI implementation must place this value in R0 or X0 when it first enters the Exception level of the caller. The context identifier can be used by the caller to point to the saved context that must be restored on a core, when it starts up at the Exception level of the caller. The caller might use other methods to implement this functionality.

5.4.2 Caller responsibilities

Before a `CPU_SUSPEND` call the Normal world must observe the following:

1. For a power down request, the calling supervisory software must have saved all the state it requires to enable resumption of operation on reset. The context saved must reflect all caller-visible state that would be lost if the affinity level indicated by the power state parameter is powered down. For more information on the required state following the application of power after a `CPU_SUSPEND` or `CPU_ON` call see Section 5.13.
2. For a power down request, the caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.
3. The caller must not assume that a power-down request will return using the specified entry point address. The power down request may not complete due for example to pending interrupts. It is also possible that, because of coordination with other cores, the actual state entered is shallower than the one requested.

4. The caller must be able to handle potential return errors:
   - `INVALID_PARAMETERS`(-2) is returned if of the following are true:
     - The power state parameter supplied is invalid.
     - The entry point address is known by the implementation to be invalid because it is in a range that is known not to be available the caller, or if a 64-bit caller provides an address with the LSB set (see Section 5.13.3).
     - The caller uses a function ID for register width that does not match the register width of the caller.
5. The caller must ensure that appropriate wake-up events are enabled to allow resumption from that state.
6. The entry point address provided by the `CPU_SUSPEND` call must be a physical address from the point of view of the caller. This means for EL1 `CPU_SUSPEND` calls trapped at EL2 the address is usually an IPA, but can be PA depending on the value of HCR_EL2.VM. For calls received in Secure world the address must be PA.
7. The context identifier is meaningful only to the caller. The value is preserved by the implementation and presented to the core when it is started up at the Exception level of the caller.

5.4.3 Implementation responsibilities: State coordination

The power state parameter expresses a constraint:

Caller allows entry down to this state, but no deeper.

For a `CPU_SUSPEND`, the primary role of the PSCI is to determine the deepest state that satisfies he constraints expressed by each core in a given affinity level.

Table 6 shows a simple example of how this arbitration would take place on a dual cluster, dual core per cluster system, that implements power gating individually per core and per cluster. The columns on the left represent the affinity level that each core has requested through the power state parameter. The columns on the right show when clusters are powered down. In this example, Affinity Level 0 is a single core, Affinity Level 1 is a cluster, and Affinity Level 2 covers the entire system.

Table 6 Example of arbitration of power down requests on a dual cluster system

Power down request affinity, by core				**Power down cluster or overall system?**		
Cluster 0		**Cluster 1**		**Cluster 0**	**Cluster 1**	**System**
Core0	**Core1**	**Core0**	**Core1**			
0	0	0	0	No	No	No
1	1	0	0	Yes	No	No
1	1	1	1	Yes	Yes	No
2	2	0	0	Yes	No	No

2	2	1	1	Yes	Yes	No
2	2	2	2	Yes	Yes	Yes

In addition to coordinating between different requests from the caller, the PSCI implementation in the Secure platform firmware might need to communicate with a Trusted OS. The method for interfacing between SPF and a Trusted OS is IMPLEMENTATION DEFINED. However the specification requires the Trusted OS to always comply with CPU_SUSPEND requests. The SPF can inform the Trusted OS that a power state needs to be entered, and the Trusted OS can use this information to take any preparatory actions. For example it might have to save its context. However, this communication must not allow the Trusted OS to modify or prevent the power state requested.

A possible mechanism for this is for the SPF to offer a notification based scheme to the Trusted OS. The notification scheme might allow the execution of Trusted OS code when the SPF receives a CPU_SUSPEND call, on processors where the Trusted OS is resident.

The Trusted OS might not be able to tolerate a particular state, for latency or other reasons. In this case, ARM recommends that the Trusted OS uses an IMPLEMENTATION DEFINED mechanism to communicate with the Normal world to ensure its constraints are considered in the power requests originating from the Normal world.

5.4.4 Implementation responsibilities: Cache and coherency management

Power-down states generally require a cache clean. Prior to powering down an affinity level, the PSCI implementation must perform a cache clean operation for all of the caches present in the affinity level. The implementation must also perform any required coherency management. In addition, the PSCI implementation needs to perform invalidation of caches on booting, unless this is automatically supported by the hardware, and to manage coherency. Sequences to be observed when powering cores up or down can be found in the Technical Reference Manuals for the relevant processors.

5.4.5 Implementation responsibilities: Call flow and save and restore

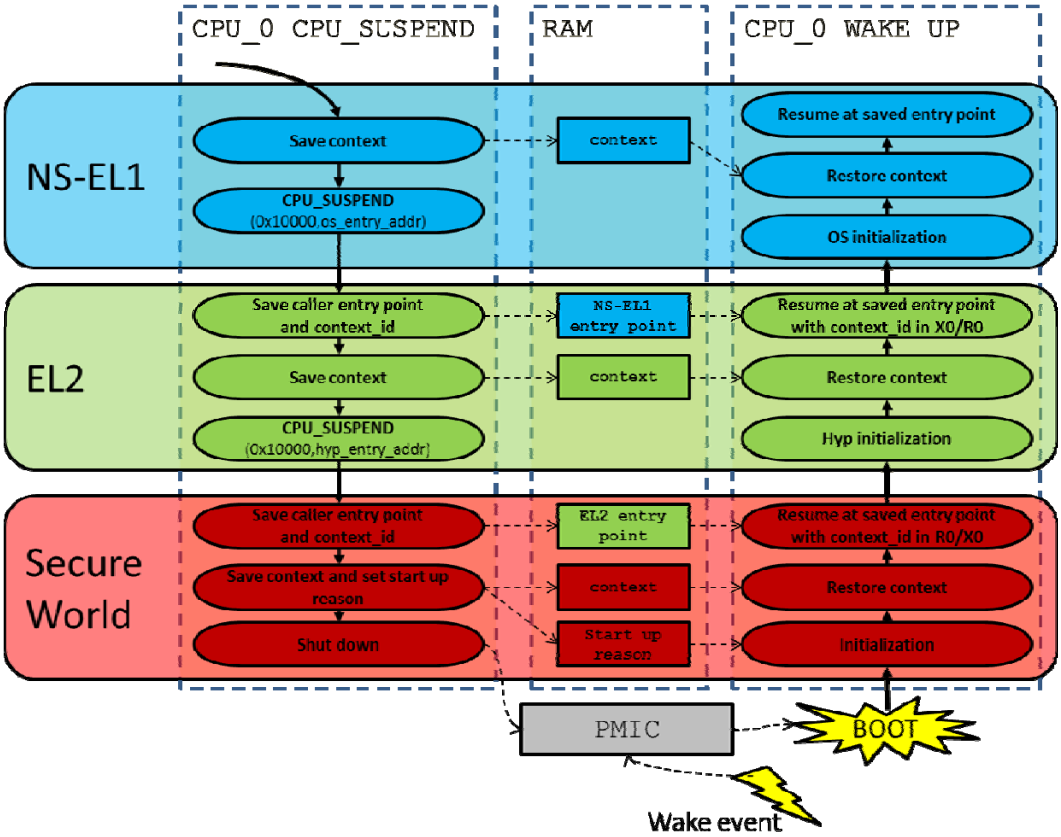

Figure 4 Example call flow for a CPU_SUSPEND call

Figure 4 shows how a `CPU_SUSPEND` call for a power down state can flow through the Exception levels and terminate in Secure platform firmware. The diagram also shows how the state is saved and restored at each stage. In the model above the hypervisor traps Non Secure-EL1 power requests. This allows a hypervisor at EL2 to save its context, the entry point and context id of the caller, before making its own power state request to the Secure world.

The Secure world works in a similar way, saving its state, the entry point and context id of its caller, and setting a start up reason, before shutting down the core. When a wake-up event occurs the power controller boots the core. On initializing, the Secure world sees the start up reason, restores context, and then re-starts the caller process at the saved entry point, supplying the saved context id. The hypervisor then can do the same with the Non-secure EL1process.

Figure 4 is an example of how the call flow might work when all Exception levels are implemented. As described in Section 5.13.1, the return Exception level that Secure world restores to is the highest Non-secure exception level implemented. This rule allows for the model described above.

In systems that implement EL2 and EL3, this rule also permits bypassing EL2, by allowing a call to go direct from Non-secure EL1 to the Secure world. However, the return path must go through EL2. How this is coordinated is determined by the hypervisor implementation. An example of this approach could be a type 2 system, which enables Non-secure EL1 calls to go direct to Secure world on the host, but traps at EL2 for guests.

Systems that do not implement EL3, and therefore do not access the Secure world, can still provide a PSCI interface to Non-secure EL1 rich operating systems that are PSCI compliant by using an HVC conduit.

5.4.6 Implementation responsibilities: State upon return

When returning from a standby state the caller should observe no change in core state, other than any timer changes expected because of the time spent in the state, and changes in CPU interface because of the wake up reason. To the core a standby state is indistinguishable from the use of a `WFI` instruction. The only exceptions are the registers used in making the SMC call that follow the SMC calling convention [4]. The return value expected in R0 or W0 is the return error code. For standby states, `SUCCESS` (0) must be returned on success. Power-down states do not return on success because restart is through the entry point address at wake-up. If not successful, the error code indicates the reason. Only the following failures are supported:

- `INVALID_PARAMETERS` (-2)

For power-down states the expected core state at wake up is identical to secondary boot state, expected when the core first arrives at the Exception level following a call to `CPU_ON`. This state is described in Section 5.13.

5.5 CPU_OFF

5.5.1 Intended use

`CPU_OFF` is designed for dynamic removal of the calling core from the system. When SPF receives a `CPU_OFF` call it must power down the calling core.

Unlike `CPU_SUSPEND`, the call is not expected to return. With this function the calling supervisory software is explicitly stating that it will no longer use a core. This situation is only reversed when a `CPU_ON` call is used to bring the core back online.

`CPU_OFF` calls can only originate from the Normal world. If the Secure world needs to manage the number of active cores, it needs to use Trusted OS specific IMPLEMENTATION DEFINED communication with the Normal world to achieve its aims. It is must not be possible for Secure world to allow a core to be seen as powered up and available by the Trusted OS, whilst the Normal world sees the core as being powered down.

5.5.2 Caller responsibilities

Before a `CPU_OFF` call the following must be observed:

1. The calling operating system must have migrated all threads and interrupts away from the core that is going to be powered down. Asynchronous wake-ups on a core that has been switched off through a PSCI `CPU_OFF` call results in an erroneous state. When this erroneous state is observed, it is IMPLEMENTATION DEFINED how the PSCI implementation reacts. Possible actions are:
    a. The PSCI implementation ignores the wake-up and keeps the core powered down.
    b. The PSCI implementation resets the system.
2. For a power down request, the caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.
3. `CPU_OFF` can have varying latency in the time it takes to shutdown the core. Depending on the states of other cores in a cluster, the call might result in core power down, or in shutting down clusters. The caller must not make any assumptions about latency.
4. A core can only power itself down.

5. In terms of coordination a call to CPU_OFF is consistent with a call to CPU_SUSPEND for power down at maximum affinity level. That is, if all the cores in a cluster call CPU_OFF, the cluster must be powered down. If only a subset of the cores call CPU_OFF, the cluster must only be powered down if the remaining processors have called CPU_SUSPEND requesting a power down state at that affinity level or a higher affinity level.

6. CPU_OFF can return the following errors:

   - DENIED (-3), as described in Section 5.9.1)

5.5.3 Implementation call flow

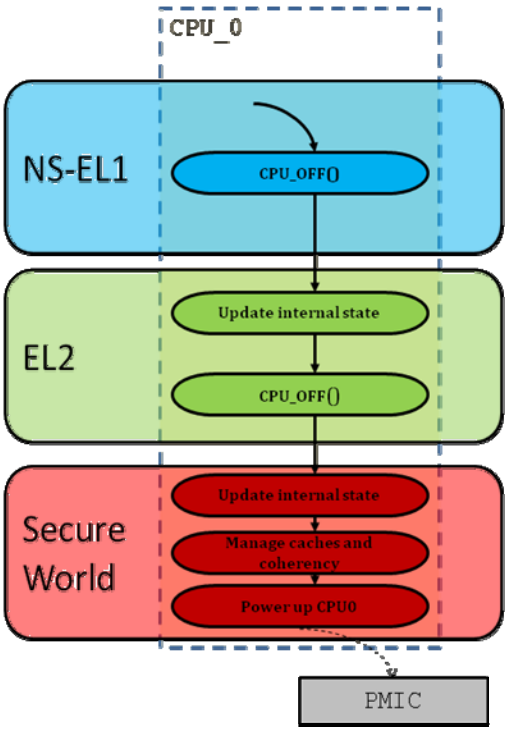

Figure 5 Example call flow for a CPU_OFF call

Figure 5 shows how a CPU_OFF can be propagated through the Exception levels in a system that implements EL2 and EL3. Whether the call is trapped at EL2 or not is defined by the hypervisor implementation.

5.5.4 Implementation responsibilities: Cache and coherency management

As with entry into a power-down state, the PSCI implementation must perform a cache clean operation for all of the caches present in the affinity level. PSCI must also perform any required coherency management. In a multicluster implementation, powering down a core typically requires:

1. Disabling the data cache to prevent data cache allocation.
2. Cleaning and invalidating any caches private to the core. This prevents any new data cache snoops or data cache maintenance operations from other cores being issued to the core that is powering down.
3. If the core is the last in a cluster, and the cluster is being powered down, clean and invalidate any shared caches private to that cluster.

4. Take the core out of coherency within that cluster. This is typically controlled by an IMPLEMENTATION DEFINED bit , for example ACTLR.SMP.
5. If the core is the last in a cluster, and the cluster is being powered down, take the cluster out of coherency.

The operations required depend on the specific cores and topology, as well as any cache coherent interconnects that may be in use. For more information see the *Technical Reference Manual*s (TRMs) of the components.

5.6 CPU_ON

5.6.1 Intended use

CPU_ON is intended for dynamic addition of cores, for use in secondary boot, hotplug, or big.LITTLE migration. When an operating system at an Exception level requires another core, it calls the `CPU_ON` function, providing an entry point address and a context identifier.

The PSCI implementation provides the necessary platform-specific code to program the power controller as required to turn on the core and then restart its execution at the entry point address. The context identifier value must be present in R0 or X0 when the core first enters the Exception level of the caller. Section 5.13 describes the expected state for a core when it accesses the entry point address. Both `CPU_ON` and `CPU_OFF` calls can only originate from the Normal world. If a Trusted OS needs to manage the number of cores, it needs to use a Trusted OS specific IMPLEMENTATION DEFINED communication with the Normal world to achieve its aims.

5.6.2 Caller responsibilities

Before a `CPU_ON` call the following must observed:

1. `CPU_ON` must implemented asynchronously. The mechanism by which supervisory software can detect that the core is finally powered up is IMPLEMENTATION DEFINED. However the provision of caller-specific entry point and context identifier parameters, allows the calling supervisory software to implement such a mechanism.
2. The entry point address provided needs to be physical from the point of view of the caller.
3. The context identifier is meaningful only to the caller. The value is preserved by the implementation and presented to the core when it starts up at the Exception level of the caller.
4. `CPU ON` can return the following errors:
    - `INVALID_PARAMETERS`(-2) can be returned if any of the following are true:
        - If `target_cpu` describes an invalid MPIDR
        - The start address is invalid. Possible causes for this are that the address is in a range that is known not to be available the caller, or if a 64-bit caller provides an address with the LSB set (see Section 5.13.3)
        - The caller uses a function ID for a register width that does not match the register width of the caller.
    - `ALREADY_ON`(-4). This error is returned if in the PSCI implementation the core is already in an `ON` state (see Section 5.6.5)
    - `ON_PENDING`(-5). This error is returned if a call to `CPU_ON` on the target core has already been made, and the core is not yet `ON` in the PSCI implementation (see Section 5.6.5)

- INTERNAL_FAILURE(-6). This error can be returned if a core cannot be powered up for physical reasons. Examples include lack of power, thermal constraints, manufacturing faults or reliability reasons.

5.6.3 Implementation call flow

Like a call to CPU_SUSPEND, a call to CPU_ON can go through the various Exception levels. Figure 6 below shows an example of a system where a hypervisor traps at EL2.

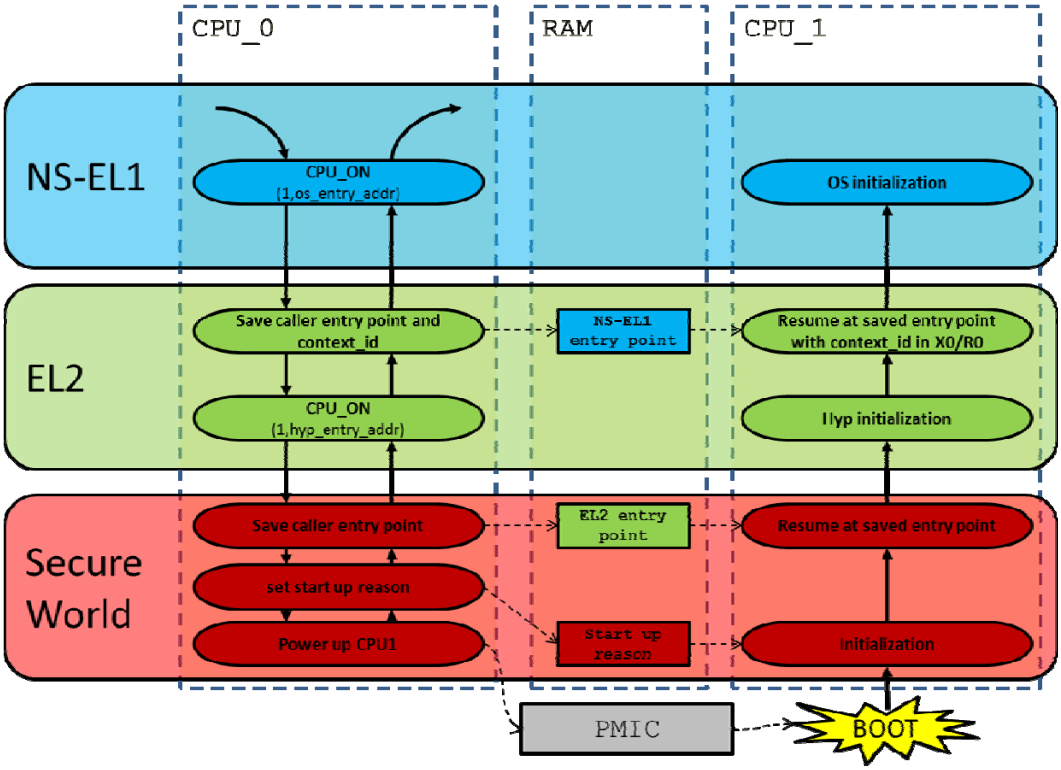

Figure 6 Example call flow for a CPU_ON call

Figure 6 is an example of how the call flow might work when all Exception levels are implemented. In the example CPU0 is requesting a power up of CPU1. As described in Section 5.13.1, the return Exception level for a CPU_ON call, is the highest Non secure exception level implemented. This rule allows for the model described above.

In systems that implement EL2 and EL3, this rule also allows bypassing EL2, that is, it enables a call to go direct from Non-secure EL1 to the Secure world. However the start-up path has to go through EL2.

5.6.4 Implementation responsibilities: Cache management

For CPU_ON the PSCI implementation must:

- Perform invalidation of caches on boot, unless this invalidation is automatically performed by the hardware.
- Manage coherency.

5.6.5 Implementation CPU_ON/CPU_OFF races

Implementation of CPU_ON and CPU_OFF requires special care to avoid races. Both the calling supervisory software and the PSCI implementation must use appropriate locks to maintain the correct state representation of a core.

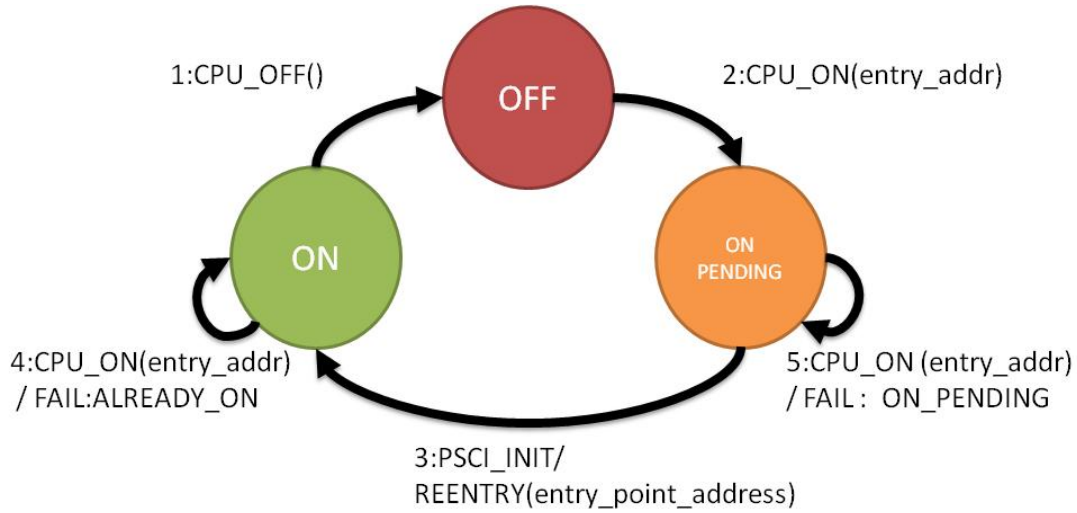

Figure 7 CPU_ON/CPU_OFF PSCI implementation state machine

The AFFINITY_INFO function defines a set of valid states for a core, see Figure 7. As seen from the PSCI implementation a core can be in any of the following states:

ON: This core has at some point been enabled with a call to CPU_ON, or is the cold boot primary core, and has not called CPU_OFF. In practice the core might be running or in a low power mode, either WFI or deeper because of a call to CPU_SUSPEND. The term ON reflects the fact that the core is still considered to be available for computation.

OFF: This core has called CPU_OFF, and the call has been processed by the PSCI implementation, or the core has not been booted yet. The core is not available for computation.

ON_PENDING: CPU_ON has been called on the target core, however it is still in the process of booting and has not transitioned to an ON state.

The state diagram also shows the valid transitions that an implementation must provide. Circles represent the states, and a state transition is represented as an arrow between two circles. The arrow label denotes the state transition such that:

1. The transition takes place while the original state is TRUE.
2. A transition can include a *post transition event*. This is indicated by the text that follows a forward slash character "/". This event happens when the destination state is TRUE.
3. After the transition the destination state is TRUE.

Figure 7 shows the following transitions:

CPU_ON (2): The implementation receives a CPU_ON call for the core in question.

CPU_OFF (1): The implementation receives a CPU_OFF call from the client running on the core in question.

PSCI_INIT (3): The core in question initializes and starts executing PSCI implementation firmware as part of the boot sequence. After this transition, when the PSCI implementation sets the core state to ON, the core re-enters the Exception level of the caller at the specified entry point address.

The diagram also describes failures to transition when invalid calls are observed. In particular:

CPU_ON (4): Fails with an ALREADY_ON (-4) error because the PSCI implementation sees the core as being ON.

CPU_ON (5): Fails with an ON_PENDING (-5) error because a valid CPU_ON call has already been made to the target core, but the core has not initialized yet, meaning it has not made transition 3 (PSCI_INIT)

An actual implementation can distinguish between further internal states. However, externally it must guarantee the states and transitions shown in Figure 7.

A caller of PSCI might maintain a similar set of states, and therefore a caller might consider any cores to be ON, OFF or ON_PENDING. The following pseudocode shows an example implementation from the perspective of the caller to CPU_ON and CPU_OFF.

```
void os_cpu_off() {
      lock(psci_state[current_cpu]);
      assert(psci_state[cpu] == ON);
      psci_state[cpu] = OFF;
      unlock(psci_state[cpu]);
      /* race window starts here*/
      CPU_OFF();

      assert(0); /* CPU_OFF() failed */
      }


int os_cpu_on(cpu,contexID) {
       int r = 0;
       lock(psci_state[cpu]);
       if (psci_state[cpu]==ON) r = E ERROR_ALREADY_ON;
       if (psci_state[cpu]==ON_PENDING) r = ERROR_ON_PENDING;

       if (r) {
             unlock(psci_state[cpu]);
             return r;
       }

       do {
             r = CPU_ON(cpu,
                        physical_addr(os_entry_point),
                        contextID);
             assert(r != ERROR_ON_PENDING) ;
             } while (r == ERROR_ALREADY_ON);

       if (!r) psci_state[cpu]=ON_PENDING;

       return r;
       }
```

This example implementation maintains a lock for each core, and uses the states described in this section, ON, OFF, ON_PENDING, but these states represent the view of the caller of the core status.

There is race opportunity, while a caller is transitioning a core to OFF (from the point of view of the caller), but before the PSCI implementation has set the state to OFF (from the point of view of the PSCI implementation). Red text above signals the start of the race window. During this time, the caller might consider the core to be OFF, while the PSCI implementation sees the core as being ON. The race closes when the PSCI implementation performs the atomic update of the state of the core to OFF.

The race is solved by the use of ALREADY_ON (-4) error. When a CPU_ON call on the same core arrives during this window the PSCI would return an ALREADY_ON (-4) error as the

implementation sees that core as still being in the ON state. A caller can use this error code to identify this situation, and retry the call. The time spent retrying will depend on the following:

1. The time between a caller setting the core state to OFF and calling CPU_OFF.
2. The time it takes the PSCI implementation to transition its representation of the core state to OFF.

Both time windows should be as short as possible. Finally when a core enters the calling exception level it can transition the state to ON:

```
void os_entry_point(){
        ...
        lock(psci_state[cpu]);
        psci_state[cpu]=ON;
        unlock(psci_state[cpu]);
        ...
        }
```

5.6.6 Implementation responsibilities: State upon return

See Section 5.13.

5.7 AFFINITY_INFO

5.7.1 Intended use

The valid states that can be returned by AFFINITY_INFO are:

ON: At least one core in the affinity level meets both of these conditions:

- The core has at some point been enabled with a call to CPU_ON, or is the cold boot primary core
- The core has not called CPU_OFF.

In practice the core may be running, or in a low power mode.

OFF: All of the cores in the affinity level have called CPU_OFF and each of these calls has been processed by the PSCI implementation.

ON_PENDING: At least one core in the affinity level is in the ON_PENDING state, as described in Section 5.5. All other cores in the affinity level are OFF.

An implementation must track the states of all the cores in an affinity level to produce the overall affinity level state. In practice, a core can be physically powered down, as a result of a call to CPU_SUSPEND, while in the ON state.

It is also possible for AFFINITY_INFO to return the DISABLED (-8) code when the compute unit described by the input parameters is valid, but is disabled for physical reasons, for example because the core is faulty.

5.7.2 Caller responsibilities

Callers must be aware that the state returned by the AFFINITY_INFO call will be affected by calls to CPU_ON and CPU_OFF, both of which could be in flight at the same time.

The caller must deal with any errors arising from the call:

- INVALID_PARAMETERS (-2) is returned if the lowest_affinity_level and target_affinity parameters describe an affinity level that is not valid. This value is also returned if the register width of the caller does not match the register width of the function ID used in the call.

- `DISABLED` (-8) is returned if the parameters describe an affinity that is disabled for physical reasons, for example, because the core is faulty.

5.8 MIGRATE

5.8.1 Intended use

The `MIGRATE` function supports powering down a core where a uniprocessor (UP) Migrate-capable Trusted OS is resident. This functionality is particularly important for multiprocessor systems, and essential for big.LITTLE systems that use a migration strategy.

Figure 8 shows the proposed migration model:

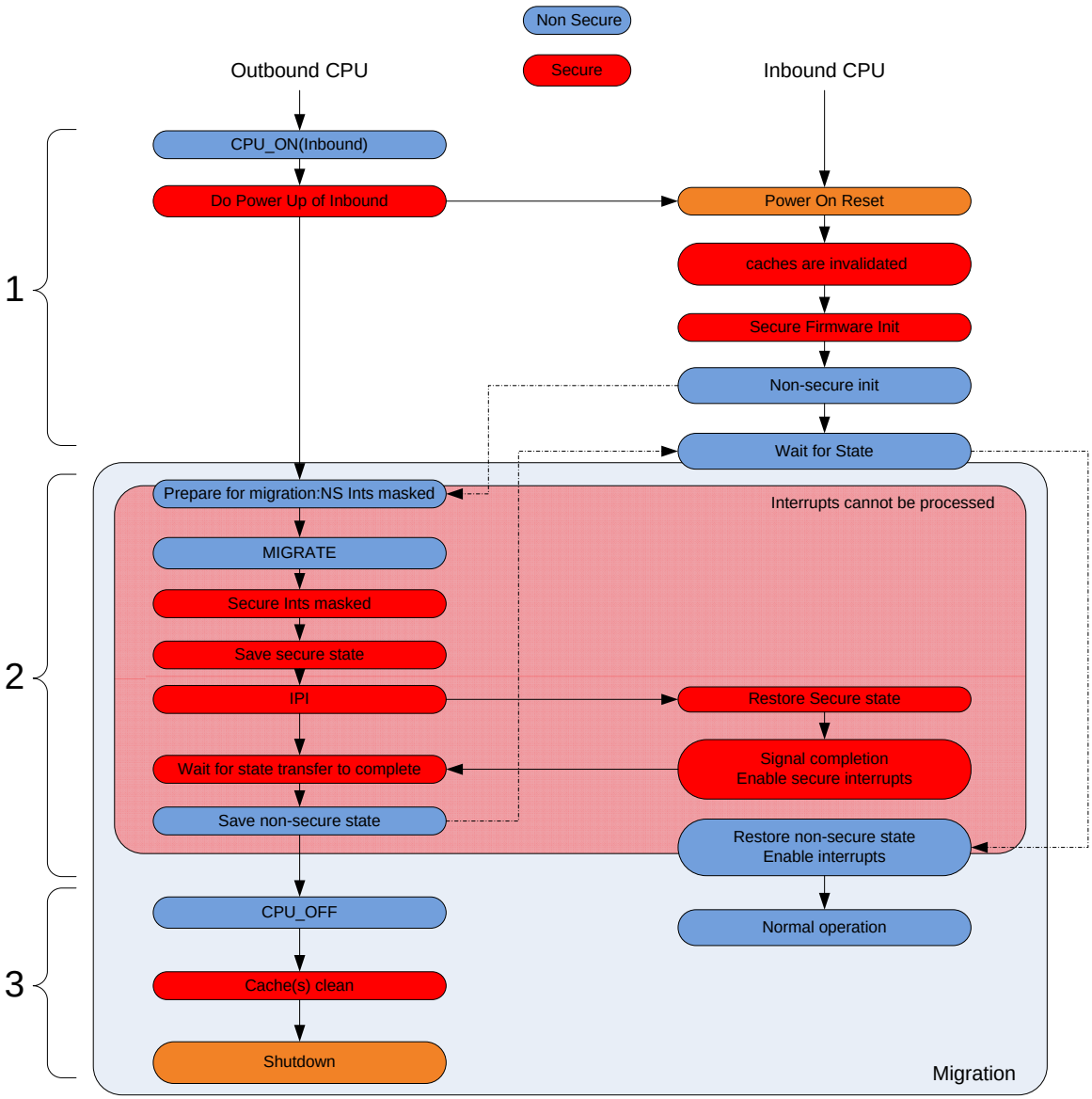

Figure 8 big.LITTLE migration

Before migration, the OSPM must request a power up for the inbound core. The OSPM could be the Rich OS or a hypervisor, depending on which OS is managing the physical migration. This is shown in the area labeled 1 in Figure 8.

When the inbound core powers up, it performs basic initialization in the Secure world, and then moves to the Normal world. The OSPM must specify an entry point address for code that places the core into a holding pattern, waiting for valid state to restore.

Ideally, the OSPM should also provide a signal so that the outbound core knows when the inbound core is ready. Up until this point there is no need to disable interrupts, and the outbound core can continue to perform useful work.

Once the inbound core is available the migration process can take place. This is shown in the area labeled 2 in Figure 8. The OSPM initiates this by masking Non-secure interrupts and issuing a MIGRATE call.

The MIGRATE call must be synchronous, and transfers Secure context from the outbound core to the inbound core. The mechanism by which this is achieved is IMPLEMENTATION DEFINED. The example in Figure 8 uses an IPI, to alert the inbound core of the presence of valid secure context. This is shown in the area labeled 2 in the diagram:

1) The outbound core masks Secure interrupts, saves Secure state, and retargets its interrupts to the inbound core.
2) The outbound core sends an IPI to the inbound core and waits.
3) The inbound core receives the IPI and restores the Secure state, unmasks Secure interrupts and then signals the outbound core.
4) The outbound core returns from the `MIGRATE` call.

If the inbound core does not respond to the IPI in a suitable time frame, or if the inbound core had not been powered up, the `INTERNAL_FAILURE` (-6) error can be returned. When the inbound side returns from the interrupt caused by the IPI it goes back to its holding pattern, waiting for valid Non-secure state to restore.

A call to `MIGRATE` must be preceded by a call to `CPU_ON`. The calling operating system can then use an IMPLEMENTATION DEFINED mechanism to determine that the core is powered up, and ready to migrate before issuing the call.

When migration of Secure context has completed, the OSPM saves Non-secure state, then releases the inbound core from its holding pattern. At this point the inbound core can restore the Non-secure context, re-enable interrupts and resume normal operation. Interrupts are only disabled whilst Secure and Non-secure context are being transferred.

The area labeled 3 in Figure 8 describes the steps the outbound core carries out once the context has migrated to the inbound core. At this point the outbound core can prepare to shut down. The shutdown does not need to happen immediately. After migration, the outbound core can call `CPU_OFF`.

With supervisory software implementing a big.LITTLE migration solution, ARM recommends that the call is made by the migration stack, or by the Non-secure Trusted OS drivers, through notifications from the migration stack. On other multicore systems ARM recommends that the call is made by the Non-secure Trusted OS driver through notifications from the hot unplug stack. This component would be supplied by the Trusted OS vendor.

Cluster migration can be implemented using this process by running it across all cores in the cluster simultaneously.

5.8.2 Caller responsibilities

The following must be observed by the caller:

1. The migration target core must be powered up before any call to `MIGRATE`.
2. The target core must not make requests for Trusted OS services until after the migration has completed.

3. MIGRATE must only be called from the core on which the Trusted OS is present. If called from another core the implementation must return a NOT_PRESENT (-7) error

4. MIGRATE can return the following errors:

- NOT_SUPPORTED (-1). If the function is not supported, or if migration is not required (see Section 5.8).
- INVALID_PARAMETERS (-2). If target_cpu describes an invalid MPIDR. This value is also returned if the register width of the caller does not match the register width of the function ID used in the call.
- DENIED (-3). This error can be returned if the Trusted OS is UP but not migrate-capable, see Section 5.8.
- INTERNAL_FAILURE (-6). IMPLEMENTATION DEFINED reason why the MIGRATE was not possible. For example the target core was not awake or did not respond within an acceptable time.
- NOT_PRESENT (-7). If called on a core where the Trusted OS is not currently resident.

5.8.3 Implementation responsibilities

Implementation of the MIGRATE function is optional.

The mechanism by which the Secure platform firmware coordinates with a Trusted OS to migrate context is IMPLEMENTATION DEFINED. However ARM suggests that a callback registration scheme is provided by the Secure platform firmware to the Trusted OS vendor. The Secure platform firmware can use this to notify the Trusted OS when key PSCI functions like MIGRATE are received.

A UP Trusted OS that is migrate-capable can register for the callbacks. The Secure platform firmware must track the resident core of a UP Trusted OS. This way it can directly respond to a number of error conditions for a MIGRATE call without having to call into the Trusted OS. In particular, it is possible to handle the following errors directly from the Secure platform firmware:

- NOT_SUPPORTED (-1)
- INVALID_PARAMETERS (-2)
- DENIED (-3)
- NOT_PRESENT (-7)

The implementation must be resistant to SMC calls for Trusted OS services from any core that is not the Trusted OS core.

5.9 MIGRATE_INFO_TYPE and MIGRATE_INFO_UP_CPU

5.9.1 Intended use

Trusted operating systems must be rigorously audited to ensure they meet their security requirements. This means that they must be kept simple. Often this leads to simple uniprocessor kernels, to avoid the inherent complexities of multicore programming. This has implications for when it is possible to hot unplug the core where the Trusted OS is resident.

The MIGRATE function provides a method to request a uniprocessor Trusted OS to move to another core. However this might not be supported by the Trusted OS vendor. The Normal world can use the MIGRATE_INFO functions to determine whether it can use CPU_OFF and MIGRATE for a given Trusted OS. MIGRATE_INFO_TYPE takes no parameters and returns:

- **Uniprocessor (UP) and migrate capable 0:** This indicates a Trusted OS that can run on only one core. The core it runs on can be changed dynamically through use of the MIGRATE function. Calls to CPU_OFF the core where the Trusted OS is currently residing return a DENIED (-3) error.
- **Uniprocessor (UP) not migrate capable 1:** This indicates a Trusted OS that does not support migration. Calls to hot-unplug the core on which the Trusted OS is running return a DENIED (-3) error. Calls to MIGRATE return the same error.
- **MP or not present 2**: This indicates that the Trusted OS is fully MP-aware and has no special requirements for migration, or a system that does not use a Trusted OS. Any core can be hot unplugged without first asking the Trusted OS to migrate to another core. In this case, calls to MIGRATE are not relevant. They return NOT_SUPPORTED (-1) and have no other effect.

In addition, MIGRATE_INFO_UP_CPU takes no parameters and returns an MPIDR-based value to indicate where the Trusted OS currently resides. The value returned uses the same format as the target_cpu parameters of CPU_ON and MIGRATE, see Section 5.1.4.

The returned value is only valid if MIGRATE_INFO_TYPE returns 0 or 1. The returned value of MIGRATE_INFO_UP_CPU is UNDEFINED if MIGRATE_INFO_TYPE returns 2.

For each possible return value of MIGRATE_INFO_TYPE, Table 7 shows the expected return values for calls to MIGRATE and CPU_OFF, when called on the CPU there the Trusted OS currently resides. The table assumes assuming that valid parameters are passed (valid target_cpu expressions)

Table 7 Return values for calls to MIGRATE and CPU_OFF (on resident core)

MIGRATE_INFO_TYPE	Return value of MIGRATE	Return value of CPU_OFF
MP or not present (2)	NOT_SUPPORTED (-1)	does not return
UP not migrate capable (1)	DENIED (-3)	DENIED(-3)
UP migrate capable (0)	SUCCESS (0)	DENIED (-3)

The information returned by MIGRATE_INFO_TYPE in must be constant and not change on subsequent calls. ARM expects that the Normal world will only use a call to MIGRATE_INFO functions once, on cold boot, and thereafter rely on the success or failure of MIGRATE calls to track the resident core of a UP migrate-capable Trusted OS.

5.9.2 Caller responsibilities

The caller can target any core with the MIGRATE_INFO functions. The caller can also assume that the value returned by MIGRATE_INFO_TYPE is constant, and will not change from one invocation to another.

The caller must avoid races between MIGRATE and MIGRATE_INFO_UP_CPU. ARM expects that the latter function will be used once by caller, upon initial boot. Thereafter, a migrating OS can make use of the success or failure of MIGRATE calls to track Trusted OS residency.

5.9.3 Implementation responsibilities

MIGRATE_INFO functions can be called from any core. How the Secure world obtains and tracks the information provided through these functions is IMPLEMENTATION DEFINED. However, ARM expects that, at cold boot, the Secure platform firmware optionally installs a Trusted OS. As part of the installation process an API can be defined between the Secure platform firmware and the Trusted OS that, in turn, can be used to provide the required return values of MIGRATE_INFO_TYPE. The Secure platform firmware will be aware of which core the Trusted OS is being installed on. Through this information, and through tracking the success or failure of MIGRATE calls, the Secure platform firmware can directly respond to MIGRATE_INFO calls from any core.

5.10 SYSTEM_OFF

5.10.1 Intended use

SYSTEM_OFF provides a shutdown API. The caller must place all cores in a known state prior to the call. Like all other APIs presented here, the calls can only originate in the Normal world. On a SYSTEM_OFF call the implementation completely removes power from highest affinity level.

If the caller is a guest running in a virtual machine this operation affects the virtual machine and might not result in any physical power changes. However, if a hypervisor is not present, or the caller is a hypervisor, the result is physical removal of power. To the caller the behavior is equivalent to a hardware power down of the whole system. Only a physical action by a user can bring the system into operation after this call. The following start must be a cold boot.

The platform can provide other IMPLEMENTATION DEFINED mechanisms to provide a shutdown. The use of PSCI for this call enables the platform firmware and Trusted OS to react to the shutdown request. However a Trusted OS must not rely on the use of any notification, and must be resistant to any sudden loss of context.

5.10.2 Implementation responsibilities

1. The implementation must support SYSTEM_OFF calls from every core in the system. If a UP Trusted OS is present, and the call does not come from the core on which it is resident, then the implementation has two options:
    a. If the Trusted OS requires it, provide an IMPLEMENTATION DEFINED mechanism to inform the Trusted OS of the impending shutdown.
    b. Use a Trusted OS that can cope with the shutdown. Particularly in mobile applications Trusted OSs must be able to deal with sudden loss of power.
2. Implementation must ensure that all cores are in a known state with caches cleaned.
3. Implementation must ensure that any data it requires is saved to non-volatile storage.

5.10.3 Caller responsibilities

The caller must perform any necessary operations to ensure a clean shutdown in its OS:

1. Ensure cores are in a known state and that any necessary data has been saved to non-volatile storage.
2. It is up to the calling supervisory software to store any information it requires on restart in storage that will survive the shutdown.

One way in which cores can be placed into a known state is to use calls to CPU_OFF on all online cores except for the last one, which instead uses SYSTEM_OFF. If a UP Trusted OS is present, this method only works if the core that calls SYSTEM_OFF is the one where the Trusted OS is resident, as calls to CPU_OFF on this core would return a DENIED (-3) error. However, the SPF and Trusted OS must ensure correctness in a way that does not depend on such a sequence being observed. Any core can call SYSTEM_OFF.

5.11 SYSTEM_RESET

5.11.1 Intended use

This function provides a method for performing a platform cold reset. To the caller the behavior is equivalent to a hardware power-cycle sequence. For a guest OS, running on a virtual machine, the SYSTEM_RESET might be virtualized. However, if there is no hypervisor present, or if a hypervisor calls SYSTEM_RESET, the whole system must be restarted, using a hardware-implemented reset.

An external debugger can use the **DBGNOPWRDWN[1,5]** signal to prevent the reset of the debug logic. This changes the effect of SYSTEM_RESET into a warm reset, as defined by the ARM debug architecture [1,5]).

5.11.2 Caller responsibilities

Where possible, cores should be placed in a known state. No specific coordination is required between cores. Except in the case when the response to the call is virtualized, when one core calls SYSTEM_RESET, the system will power cycle.

5.11.3 Implementation responsibilities

See Section 5.11.1.

5.12 Discoverability

The functions defined here are independent of the conduit used to invoke them. The PSCI specification expects that, at boot time, a hypervisor and Rich OS can discover which conduit applies. The ARM architecture provide feature registers in the ARMv7 and ARMv8 System registers, that can be used to determine at runtime whether EL2 and EL3 are implemented. This, in theory, could be used to determine the conduit, SMC or HVC. However, this does not determine whether firmware support for the PSCI interface is present. Therefore, ARM proposes that device implementers provide firmware tables, for example FDT or ACPI, that describe whether the interface is present.

In addition, ARM recommends that the tables provide:

- An entry for each individual function, with absent entries indicating functions that are not implemented.
- An interface version section, that not only allows discovery but also allows changes over time.
- Conduit information: SMC or HVC.

ARM recommends that the functions are described using function IDs in the standard call range defined in the SMC calling conventions [4]. However the use of firmware tables gives some flexibility when this is not possible.

The following is an example device tree binding for a PSCI implementation:

```
 psci {
      compatible             = "arm,psci-0.2", "arm,psci";
      method                 = "smc";
      psci_version           = <0x84000000>;
      cpu_suspend            = <0x84000001>;
      cpu_off                = <0x84000002>;
      cpu_on                 = <0x84000003>;
      affinity_info          = <0x84000004>;
      migrate                = <0x84000005>;
      migrate_info_type      = <0x84000006>;
      migrate_info_up_cpu    = <0x84000007>;
      system_off             = <0x84000008>;
      system_reset           = <0x84000009>;
};
```

This provides version information and the conduit. For each function it provides the call UID required by the SMC to identify the function. In this example all the calls in the current version of the interface are supported. This example uses 32-bit identifiers. The firmware tables should provide descriptions appropriate to the register width of the operating

systems that will be reading them. In this way, a 64-bit kernel can use the 64-bit identifiers, and a 32-bit kernel can use the 32-bit identifiers. If a call is not present the implementation must not describe it.

ARM proposes that Secure firmware provides descriptions of the topology of the system that supervisory software can access to determine:

- The hierarchy of clusters available in the system.
- The composition of each cluster, that is, what cores are in each cluster.

Also it is desirable for the firmware tables to express the power states available at each level of the hierarchy. This would describe the available power state parameters that can be used in `CPU_SUSPEND` calls.

5.13 Initial state after CPU_ON, CPU_SUSPEND

The following sections describe the expected state of a core or affinity level when a core starts up, following a call to `CPU_ON`, or on wake-up from a power-down state, following a `CPU_SUSPEND` call.

5.13.1 First Non-secure exception level

When returning from a `CPU_SUSPEND` call after a power-down, or following a `CPU_ON` call, the first Non-secure Exception level that the Secure world must move the core to is the highest implemented Non-secure Exception level. This is described here as EL_{NS}.

In a system that implements EL3 and EL2, it is not possible to use any services of a hypervisor until the Secure world has enabled use of the `HVC` instruction. This is controlled by the Secure configuration register HCE field, SCR.HCE or SCR_EL3.HCE. Until the hypervisor has been enabled in this way, the highest Non-secure Exception level is EL1. When the hypervisor has been enabled, EL_{NS} is EL2.

For the entry to EL_{NS}:

- If EL_{NS} is using AArch32 the initialization code must set the processor mode to be entered.
- If EL_{NS} is using AArch64, the initialization code must set the stack pointer that will be used. With PSCI the stack pointer must be the one associated with $EL_{NS,}$ that is, SP_EL_{NS}, denoted as EL2h or EL1h.

The following table shows this requirement:

Table 8 First Non-secure Exception level and state on entry

			Caller using AArch32		Caller using AArch64	
	Caller	Callee	EL_{NS}	Processor mode	EL_{NS}	Stack pointer
Hypervisor Installed	EL1	SPF	EL2	Hyp	EL2	EL2h
	EL2	SPF	EL2	Hyp	EL2	EL2h
Hypervisor not installed	EL1	SPF	EL1	Svc	EL1	EL1h

When control passes from EL2 to EL1 the startup state must be:

- Supervisor mode, if EL1 is using AArch32.

- The EL1h stack pointer state, if EL1 is using AArch64.

5.13.2 Entry point address

The entry point address parameter passed in calls to `CPU_ON` or `CPU_SUSPEND` must be a valid physical address at EL_{NS}. For systems using virtualization, this might mean that PSCI calls from Non-secure EL1 have to be trapped to EL2. This applies when the physical OSPM as defined in Section 3.6 resides at EL1. There are two possible schemes:

1. Calls from EL1 bypass EL2 altogether, and are routed directly to the Secure platform firmware. In this case, the OS at EL1 that is making the PSCI calls must either:
   - Ensure that stage 2 address translation is disabled.

     Note: Stage 2 address translation is controlled from EL2, by HCR.VM or HCR_EL2.VM. The mechanism by which an OS at EL1 can determine whether Stage 2 translation is disabled is IMPLEMENTATION DEFINED
   - Use an IMPLEMENTATION DEFINED mechanism to obtain a valid physical address for the entry point. In this case, the hypervisor at EL2 and the OS at EL1 must have an IMPLEMENTATION DEFINED mechanism to determine the address at which EL1 resumes execution.
2. EL1 calls are trapped by EL2, either by setting the HCR.TSC or HCR_EL2.TSC bit to 1 or by enforcing use of an HVC conduit. EL2, knowing that the caller contains the physical OSPM, forwards the PSCI call to the Secure platform firmware using an appropriate entry point address

5.13.3 Initial core configuration

For an Exception level to start execution, higher Exception levels must perform appropriate initialization. From ARMv8, platform reset values are defined to provide the minimal functional set for the highest implemented Exception level to start execution. This Exception level then has to set up enough state for a lower Exception level to execute.

This section describes the items that must be correctly initialized when a core starts up in response to a power-down state wake-up for **`CPU_SUSPEND`**, or through a call to **`CPU_ON`**.

Execution state on ARMv8 systems

The ARMv8 architecture defines two execution states [5], AArch32 and AArch64. On an ARMv8 system, the Execution state entered when starting up a lower Exception level must match the Execution state in use by that Exception level when the PSCI call is made. **Table 9** shows this requirement.

Table 9 Required Execution state on Exception level start up

Implemented Exception levels	Caller	Callee	Transition	Required Execution state
EL1, EL2, and EL3	EL1	EL3	EL3->EL2	EL2 Execution state at time of call
			EL2->EL1	EL1 Execution state at time of call
	EL1	EL2	EL2->EL1	EL1 Execution state at time of call
EL1 and EL2	EL1	EL2	EL2->EL1	EL1 Execution state at time of call

EL1 and EL3	EL1	EL3	EL3->EL1	EL1 Execution state at time of call

Endianness

The caller endianness must be restored when starting up the core at the Exception level from which the call was made. This means, for the Execution level where execution is restarting:

- If starting in AArch32 state CPSR.E must be set to the appropriate value.
- If starting in AArch64 state SCTLR_ELx.EE must be set to the appropriate value. ELx is the Exception level being returned to.

Note that only the highest implemented Exception level has a defined reset value for SCTLR_ELx.EE.

Exceptions

The appropriate asynchronous exception masks must be set when starting up the core at the Exception level from which the call was made, or at EL_{NS}. Typically, this means that for the Exception level where execution is restarting:

- If starting in AArch32 state, the CPSR.{A,I,F} bits should be set to {1,1,1}.

Note: On an ARMv7 implementation that includes EL3 (the Security Extensions) but does not include EL2 (the Virtualization Extensions), setting mask bits to 1 in the CPSR masks the interrupts for all Security states and processor modes, regardless of the values of SCR.{FIQ, IRQ, EA, FW AW}. For these implementations, if Secure platform firmware, or the Trusted OS, requires FIQs and asynchronous aborts to be routed to the Secure world, the Secure platform firmware must start Non-secure EL1 with CPSR.{A, F} set to {0, 0}

- If starting in AArch64 state, the SPSR_ELx.{D,A,I,F} bits must be set to {1, 1, 1, 1} . ELx is the Exception level being returned to.

MMU, Cache and branch predictor enables

When starting up the core at the Exception level from which the call was made, execution must start with the stage 1 MMU and caches disabled. Also, if starting up in an Exception level that is using AArch32, Branch predictors must be disabled. This means:

- If starting in AArch32 state, the SCTLR.{I, C, M} bits must be set to {0, 0, 0}. In addition the Z bit must set in a way that is consistent with the reset behavior of the platform.
- If starting in AArch64 state, the SCTLR_ELx.{ I, C, M} bits must be set to {0, 0, 0}. ELx is the Exception level being returned to.

On start-up, from the point of view of the caller the caches must be clean and coherent with main memory.

T32 support

If the caller is in AArch32 state, and the entry point address specified in a call to `CPU_ON` or `CPU_SUSPEND` has bit[0] set to 1, the core must be started in T32 state at the Exception level of the caller. In this case the entry point address specified by the caller is equivalent to the address passed in the call, but with bit[0] set to 0. In AArch64 state, if a 64-bit caller passes a entry point address with bit[0] set, the call should return `INVALID_PARAMETERS` (-2).

Generic Timer

CNTFREQ holds the frequency of the Generic timer. This register must be initialized by the Secure world, to enable use of the Generic timer.

Context ID

When the core first enters the Exception level of the caller, the context ID value supplied in the CPU_SUSPEND or CPU_ON call must be present in:

- X0 if the caller was using AArch64
- R0 if the caller was using AArch32.

6 Changes from first proposal

This version of the specification has the following functional changes from the first draft proposal version of PSCI:

1. Provided values for Function ID parameter of all functions.
2. Added context ID parameters to `CPU_ON` and `CPU_SUSPEND`.
3. Removed `DENIED` (-3) return value of `CPU_SUSPEND`.
4. Removed `power_state` parameter for `CPU_OFF`.
5. Removed `INVALID_PARAMETERS` (-2) for `CPU_OFF`.
6. Specified format for `target_cpu` parameter of a `CPU_ON` to be based on MPIDR.
7. Removed `DENIED` (-3) return value of `CPU_ON`.
8. Added `ALREADY_ON` (-4) return value of `CPU_ON`.
9. Added `ON_PENDING` (-5) return value of `CPU_ON`.
10. Added `INTERNAL_FAILURE` (-6) return value of `CPU_ON` and `MIGRATE`.
11. Added NOT_PRESENT(-7) return value to MIGRATE
12. Renamed -3 error code from “Core Not Available” to `DENIED` for `MIGRATE`.
13. Added the following functions:
    a. `PSCI_VERSION`
    b. `AFFINITY_INFO`
    c. `MIGRATE_INFO_TYPE`
    d. `MIGRATE_INFO_UP_CPU`
    e. `SYSTEM_OFF`
    f. `SYSTEM_RESET`
14. This revision defines a version number for PSCI, currently 0.2
15. Made all functions except `MIGRATE` and `MIGRATE_INFO_UP_CPU` compulsory, and consequently removed `NOT_SUPPORTED` (-1) return code from `CPU_ON`, `CPU_SUSPEND`, and `CPU_OFF`.

7 Glossary

The following table describes some of the terms used in this document.

Table 10 Glossary terms

Term	Description
ACPI	The Advanced Configuration and Power Interface specification. This defines a standard for device configuration and power management by an operating system.
FDT	Flattened Device Tree. This is a hardware description standard. Firmware tables are constructed that describe the hardware. These tables are passed to the operating system at boot time. An operating system can interrogate the data they contain when it needs to discover the hardware properties of a device.
HVC	The Hypervisor Call instruction, or the associated exception. This requests a hypervisor function, causing the core to enter EL2.
MP (Multi-Processor)	Refers to a software stack, typically an operating system, that supports simultaneous operation across multiple cores.
Normal World	The execution environment when the core is in the Non-secure state.
OSPM	Operating System-directed Power Management. Typically, this acronym refers to the components of an operating system that provide power management for the platform. In this document OSPM refers to software components that are involved in the selection and application of power states for individual cores or clusters, or overall system states.
Rich OS	Application operating system such as Linux or Windows.
Secure Platform Firmware (SPF)	Owned by the silicon vendor and OEM. This firmware layer is the first thing that runs at boot on an application processor. It provides a number of services, including platform initialization, the installation of the Trusted OS, and routing of Secure Monitor Calls. Some calls are destined for the SPF and some are destined for the Trusted OS. SPF can run in EL3 and secure EL1 on ARMv8 systems using AArch64 in EL3. For ARMv7 systems, or ARMv8 systems using AArch32 at EL3, SPF executes in EL3. The SPF must include the implementation that acts on power management requests expressed by the Power State Coordination Interface.
Secure World	The execution environment when the core is in the Secure state. When the core is executing in EL3 it is in Secure state.
SMC	The Secure Monitor Call instruction, or the associated exception. This requests a Secure Monitor function, causing the core to enter EL3.
TEE	Trusted Execution Environment. An environment that runs alongside but isolated from other execution environments, and that, because of the rules of the environment, can be trusted in clearly-defined ways. To achieve this, a TEE has security capabilities and meets certain security-related requirements.
Trusted OS	This is the operating system running in the Secure world. It supports the TEE.
UP (Uniprocessor)	Refers to a software stack, typically an operating system, that does not support simultaneous operation across multiple cores.